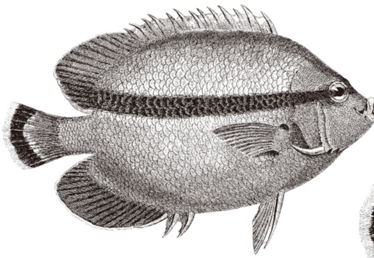
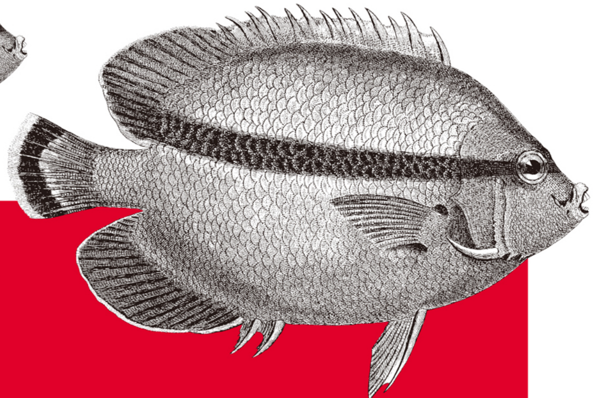
O'REILLY

TURING 图灵程序设计丛书





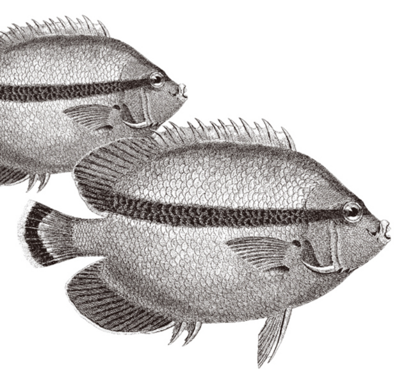
MySQL与MariaDB学习指南

Learning MySQL and MariaDB

MySQL与MariaDB之父Monty Widenius作序推荐

领你走上使用数据库的正确之路，助你迈入专家行列





[美] Russell J.T. Dyer 著

袁志鹏 译

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

MySQL与MariaDB学习指南

Learning MySQL and MariaDB

［美］Russell J.T. Dyer 著

袁志鹏 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北　　京

图书在版编目（CIP）数据

MySQL与MariaDB学习指南 / (美) 罗素·戴尔 (Russell J. T. Dyer) 著 ; 袁志鹏译. -- 北京 : 人民邮电出版社, 2016.10
（图灵程序设计丛书）
ISBN 978-7-115-43571-2

Ⅰ. ①M… Ⅱ. ①罗… ②袁… Ⅲ. ①SQL语言－指南 ②关系数据库系统－指南 Ⅳ. ①TP311.132.3-62 ②TP311.138-62

中国版本图书馆CIP数据核字(2016)第227178号

内 容 提 要

本书使读者不仅能够深入了解 MySQL 这种主流数据库，还能全面掌握开源数据库新秀 MariaDB 的使用方法。书中内容由浅至深、层层深入，从分步介绍如何安装 MySQL 和 MariaDB，到以虚构的观鸟网站为例，详解数据库的各种操作。具体内容包括：数据库的结构；数据的插入、选取、更新、删除、连接和子查询；字符串函数、日期和时间函数、聚合函数与数值函数等。最后一个部分从更高的角度介绍数据库的管理，内容涉及用户账号及权限、数据库的备份与恢复，以及利用应用编程接口结合 C、Perl、PHP、Python、Ruby 等不同语言与数据库交互。

本书面向想要从头开始学习并快速掌握数据库核心知识与实践方法的读者。

◆ 著　　[美] Russell J.T. Dyer
译　　袁志鹏
责任编辑　朱　巍
执行编辑　谢婷婷　吴威娜
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：19.25
字数：460千字　　　　2016年 10 月第 1 版
印数：1－2 500册　　　　2016年 10 月北京第 1 次印刷
著作权合同登记号　图字：01-2015-5421号

定价：79.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第三部分 数据处理基础

第四部分 内置函数

第五部分 数据库管理

序

在你阅读书中 MySQL 和 MariaDB 的内容之前，我打算先讲讲，我们分别在大约 20 年前和 5 年前创建 MySQL 和 MariaDB 的目的，以及这两个数据库系统的现状与我对它们未来的期望。我认为，这会有助于你了解它们。顺便，为了给你打气，我想告诉你，MySQL 和 MariaDB 是长盛不衰的，你对它们的钻研以及你在此书上花费的精力，都将令你受用良久。

MySQL的起源

我和我的商业伙伴 David Axmark 之所以会创造出 MySQL，是因为那个年代没有什么好用的、免费的开源数据库系统。我们当时只是创建了一个类似 mSQL 的数据库，它不是开源的。但这个数据库启发我们为客户创造出一个新的数据库系统，这就是后来的 MySQL。对于这个 MySQL 的雏形，我们并没有什么宏大的开发计划，只是要满足客户的需求。我们不断地学习、发现，并根据实际需求进行开发，作为本书的读者以及 MySQL 和 MariaDB 的入门者，你可能也在这样做。

创造好之后没多久，我们就发现有不少机构都有类似的需求。既然我们已经开发好了这个数据库，便决定将其对外开放，并给其取名为 MySQL。

我们这样做的动机之一是觉得这东西挺有用的，值得拿来回馈开源社区（当时很多开源项目都是没什么用的）。我们希望这个世界变得更美好一点——当时我们真不知道 MySQL 会有现在这么大的影响力。同时，我们也希望将 MySQL 公之于众能够带来收益，以便资助 MySQL 的长期开发。当然，我们也想通过 MySQL 来致富。因为觉得这东西应该是有前途的，所以我们全身心地投入进去。而事实上，结果是我们为这个世界作出了很大贡献，甚至远远超出我们的想象。

如今，世界上 80% 以上的网站都在用 MySQL，可以说 MySQL 推动了互联网以及由互联网而生的一切事物的发展。其影响力是不可估量的。如果没有免费又可靠的 MySQL，许多成功的（包括现在一些大型的）网站和企业，可能根本无法诞生。因为在当时，很多创始人和创业公司都没钱创建网站。商业数据库软件的价格不菲，一些最有创造性的网络组织，如谷歌、维基百科和 Facebook，都难以跨过这一障碍。另外，商业数据库还有其他

缺点。例如，互联网公司对访问性能有很高要求，但商业数据库却对这方面毫不关注。此外，商业数据库需要专门的开发人员来使用和管理，而这些人的工资要求也相当之高。

正因如此，MySQL 非常符合创业公司的要求，可以帮助它们成为互联网的重要组成部分，以及大多数人的日常所需。MySQL 曾经是而且目前依然是互联网发展中的一个关键要素，而且这在未来也不会改变，因为 MySQL 的使用量依然在增长，而 MariaDB 则势头更劲。那些认为新的数据库或 NoSQL 会令 MySQL 跌价的人，终将发现自己站错队伍。

由于 MySQL 称霸已久，无法撼动，而且人们总是习惯使用熟悉的东西，所以就算有更好的东西出现，可能也难以取而代之。若真要取代 MySQL 开源数据库的霸主地位，除了要有更好的功能，还要允许用户运用现有的知识轻松地迁移数据。而 MariaDB 就是在 MySQL 的基础上，拥有更多的功能和潜能，因此，我们认为 MariaDB 就是 MySQL 的替代者。

MySQL和MariaDB的现状

当然，MySQL 和 MariaDB 都不是完美的。事实上，任何数据库都不是完美的，但对大多数人来说它俩已足够优秀。我们既要方便网站的开发，又要提供良好的性能。所以，我们采用多线程技术，这使我们在负载方面优于很多同行。此外，我们自始至终采用最先进的技术，努力兼容新硬件并优化各种常用软件和部署方式。我们在软件改进上精益求精，可以每个月发布社区版的一个小更新，每年公开一个新版本。而这也表明，我们的社区运作正常，稳步发展。

所有学习和想要使用 MySQL 和 MariaDB 的人，都不用担心接口的改变，因为我们开发的数据库一直在适应环境的改变，无论时代怎样变迁，你要的功能永远都在。这也确实是我们令人信赖的一点。有些软件时不时搞一些标新立异的功能，甚至过一两年就换一个全新的系统，而这在 MySQL 和 MariaDB 身上是不会发生的。

之前我们讲过，要取代 MySQL 这个霸主是不容易的，所以我们尽量让 MariaDB 贴近 MySQL，让你无痛迁移。另外，我们还在（并将在）MariaDB 上加入很多好用的功能，以便你获得更好的使用体验。所以，MariaDB 也是一个不错的选择。

不只是服务器

除了用于网站开发，嵌入其他软件后，MySQL 和 MariaDB 还可用于单独的应用程序。尤其是嵌入式方面，它们近来的增长幅度比以往任何时候都要大。另外，由于现在流行使用云服务器，而在云中部署商业数据库十分昂贵，所以，MySQL 和 MariaDB 在这方面也成为了大众的首选。

同样，在移动开发方面，它们也是最佳方案。无论你的应用是部署于云端还是自建服务器，依靠 MySQL 和 MariaDB 出众的扩展性，你将无惧因为移动设备普及而带来的爆发式访问增长（事实上有些网站的移动端访问甚至多于桌面端）。此外，如果你用的是 MariaDB 10.1，那么它新引入的加密功能将使应用的安全性更上一层楼，而这是很多其他数据库产品不能提供的。

MariaDB：差异和希望

我对 MariaDB 是充满希望的，我正在 MariaDB 基金会工作，致力于拉拢其他公司协同开发。这在 MySQL 的发展过程中是缺失的，所以，MySQL 无法满足全世界，无法满足未来。但我们希望 MariaDB 能做到，这就需要更多公司的合作。我们十分开心看到谷歌的参与，同时还希望更多这样的大公司参与进来。此外，光有大众的支持也是不够的，像自由与开源软件基金会，它有很多公司参与开发，但其中没有负责协调工作的。我期望 MariaDB 基金会能起到协调的作用，令大家的努力都能用到点上。当然，这反过来也是对大家有益的。而对于 Oracle 控制下的 MySQL，就不用在这方面指望太多了。Oracle 可没有保证会在未来继续开放 MySQL 的代码。而 MariaDB 则不同，它是由始至终、全心全意地开源的。所以很明显，MariaDB 比 MySQL 更加遵守开源规则，更加贴近多数人的想法。

MariaDB 基金会的作用，就是确保 MariaDB 的开发过程公平、公开、公正。它将保证 MariaDB 始终开源，这是它的首要职责。MariaDB 基金会的另一个职责是确保所有想要开发 MariaDB 的公司都以平等的方式参与开发。如果有公司为 MariaDB 提供了补丁，他们可以进行提交，而下一版的 MariaDB 将包含该补丁。这是很多自称开源的软件项目做不到的，包括 MySQL（当你想给它提交补丁时，Oracle 极有可能是不理你的）。而 MariaDB 的包容则天性使然，我们乐意接受任何意见。

举个例子，假设 MariaDB 的竞争对手 Percona 想给 MariaDB 提交一个补丁，以使其后台软件 XtraBackup 能运行得更好。当然，MariaDB 是不想帮助竞争对手的，但是，这也不由它话事。只要基金会认可那个补丁，那么就会加上，因为基金会是只看技术，不考虑商业因素的。

一个开源软件要想存活，就必须能够解决实际问题。虽然 MySQL 一开始有各种不足，包括功能少，但它一直聆听大众的需求，因此才得以成为数据库界的黑马，冲出重围。这一点，我们在 MariaDB 上做得更加努力。尽管 MariaDB 也不可能完全符合所有人的要求，但它会比别的产品更好。

MySQL和MariaDB的未来

如果你将来打算从事 MariaDB 相关的工作，将能享受到我们牛人齐集的开发团队的全力的、长期的支持。

拿近来发布的 MariaDB 10.1 来说，它之所以兼容 Galera 集群（用于管理并行数据库）这一功能，正是为了更好地支持新的加密功能。为什么要这样大费周章呢？因为在过去的几个月里，一些政府部门和大企业遭受了黑客攻击，致使人们对软件行业的安全问题十分担忧。而更好的加密技术，将使数据更难被窃取。MariaDB 的这一努力就是为了改变人们认为开源软件安全性不佳的印象，并证明自己是紧跟时代的。虽然如今不少商业数据库开发商都误导人们，说 MySQL 和 MariaDB 不够安全，从而骗取更多生意，但是我相信，只要你用过 MariaDB 10.1，就会明白那是谣言，同时知道 MariaDB 是优于 MySQL 的。

如果你对编译好的商业数据库是否留有后门抱有怀疑，或者担心开源软件是否安全，那么，最直接的办法就是去检查我们提供的开源代码。因为我们的代码非常坦白，所以我

想，有以上担忧的组织或国家，在未来应该都会成为我们快速扩展的市场。而且，说来讽刺，事实上越是遮遮掩掩的政府和组织，越应该喜欢开源软件，因为开源软件的仇家比商业软件的要少，更不容易遭受攻击和破解。

学习MySQL和MariaDB的前途

MySQL 和 MariaDB 都兼容 SQL，它是一种有大约 30 年历史的编程语言。虽然 SQL 一直没多大改变，但它功能强大，随处可见。只要你掌握好某一套 SQL 数据库，那么想迁移到另一种，是没什么难度的。也就是说，学习 MySQL 或 MariaDB，对你的数据库开发和管理生涯有百利而无一害。现在还没有任何迹象表明 MySQL 或 MariaDB 会在 50 年后消失，事实上，MySQL 在过去 20 年提出的所有概念，在今天甚至未来几十年，都不会失色。唯一的变化是会加入新功能，以便人们完成特殊任务。而本书中介绍的通用技能，将使你终身受益。

有关学习MySQL和MariaDB的建议

想学好 MySQL 和 MariaDB，只读本书是不够的，你还得安装 MySQL 或 MariaDB，执行书中的示例，并完成每章后面的习题。此外，还要应用书中提到的 SQL 语句、函数和工具来做些自己的玩意。如果不去实践的话，那么你看过的东西都会忘记。如果你想不出要搞些什么玩意，可以试试用 MySQL 或 MariaDB 来创建一个网站，并尽量去解决各种数据库相关的问题。只要你持之以恒地操练自己，就会学到更多。不断实践，才能将基础打牢。

除此之外，你还可以通过加入论坛、邮件列表和 IRC 来学习更多知识，在社区积攒人气，以及开发出商业网站（这甚至可以助你找到好工作）。用所学的知识帮助别人，你不仅会受人欢迎，还会因为解释各种概念而理解得更加深刻。

——Monty Widenius

2015 年 1 月于西班牙马拉加

献给我的母亲 Fortunata Serio，是她给了我生命，并教我向善、有爱，以及说话——不会说话可当不了作家。

也献给我的继父 Andrew Gambos，他一直默默地支持我，并教会我如何自力更生。

前言

MySQL 是当今最流行的开源数据库，高效且稳定，备受公众网站的青睐。即使你对它不熟悉，也可能天天都在跟它打交道。当你登录谷歌、亚马逊、Facebook 和维基百科等知名网站时，就会用到 MySQL。不仅许多大型网站用它保存数据，数之不尽的小网站也在用着它。此外，很多非网络应用也采用 MySQL 作为数据库。在需要时，它可以发挥快速、稳定和小巧的优点。

1995 年，MySQL 由 Michael“Monty”Widenius 和 David Axmark 创造，并使用 GNU 通用公共授权。当年他们在瑞典创立了 MySQL Ab（Ab 是瑞典语中的有限公司或股份公司），而该公司几年后又成了美国的 MySQL Inc.（英语 incorporated 的缩写）。直到 2008 年 1 月，它被 Sun 公司收购。尽管 Sun 说“我们绝对会大力发展 MySQL”，但在 2009 年 4 月，它自身却被 Oracle 收购了。因为 Oracle 是卖闭源数据库的，可以说是 MySQL 的一大竞争对手，所以当时很多人担心 MySQL 这个改变世界的开源软件会就此被扼杀。不过事实证明，在收购五年之后，这种情况都没有发生。我们可以看到，现在 MySQL 的功能更加丰富了，而其相关的开发者（无论是在 Oracle 内部还是外部）数量也增加了不少。

出于对 Oracle 收购的不爽，Monty 又开了一家新公司——Monty Program Ab，它在 MySQL 的基础上开发出了一个分支——MariaDB。[1] 因为 MySQL 是遵循 GPL 协议的，所以使用它或往它身上添加东西都是免费和合法的。与此同时，MySQL Inc. 服务部的前高级副总裁 Ulf Sandberg，以及 MySQL 的一班老员工，离开 Sun 和 Oracle，建立了 SkySQL Ab，为 MySQL 和 MariaDB 用户提供支持、咨询和培训等服务。2013 年 10 月，Monty Program Ab 合并到 SkySQL Ab 中，该公司在 2014 年 10 月改名为 MariaDB Ab。而 MariaDB 的授权认证则由 MariaDB 基金会控管，而不是 Oracle 或其他公司。

注 1：顺便提一句，MySQL 的这个名字来源于 Monty Widenius 的第一个女儿，My Widenius。而 MariaDB 则源于其第二个女儿，Maria Widenius。

现在，有些相关的社区，因为不想与大型专利软件公司扯上关系，所以也将数据迁移到了MariaDB。此外，不少操作系统发行版、硬件和软件包，也把自带的数据库改为MariaDB（可能带上MySQL，或完全不带）。因为替换并不困难，应用层不需修改任何代码（当然，如果你想使用MariaDB有而MySQL没有的功能，那就需要加入新的调用命令），所以许多网站也愿意追随这一潮流。

尽管所有权、公司名，甚至软件名经历多次变更，但这个软件在社区中相传近30年的精神与内涵却没有丢失。

MySQL和MariaDB简单易懂，没有什么学习门槛。如果你想从头开始，并快速形成生产力，那么本书可以作为你的入门书。当然，对于稍有基础但未得要领的同学，本书也是非常合适的。因为MySQL和MariaDB的基本操作是一样的，所以这本入门书的内容同时适用于两者，你可将文中的MySQL理解为MariaDB，反之亦可。

阅读方法

一般来说，你应按本书的章节编排顺序读下去。但这并不是说你不能跳过某些章节。尤其是，很多人都会跳过第一部分。要是你已装好MySQL，那么就可以不用看第1章（导论部分）和第2章（讲述如何安装MySQL和MariaDB）了。要是你从未用过MySQL，就不应该略过第3章。而之后的章节，除了第五部分（这部分的章节与管理技术有关，并非所有人马上都能用到），你都应该按顺序学习到底。

大多数章节最后都有一套练习题，它们可以帮助你思考之前读到的内容。通过完成这些习题，你会巩固在该章示例中学到的知识。另外，尝试练习所有章节中给出的示例，你会受益良多。各章后面的习题，就算不以前一章的知识为基础，也是以前面的章节为基础，所以你需要学好前面的知识，才能顺利完成它们。

文字界面与操作系统

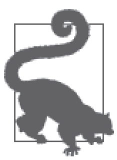
因为Windows的风行，导致很多人都以为，用GUI（图形用户界面）来操作复杂的软件或系统是最快捷的做法。确实，一图胜千言，但如果你不是要表达很复杂的东西，是不需要图形的。换句话说，如果你只对数据库进行一些很小的改动，是不需要用GUI的。

尤其是，我不喜欢用GUI来操作MySQL或服务器，因为GUI每次更新换代都会变样，而命令行却十年如一日，处处通用。如果你知道如何从命令行配置服务器，那么就能轻松应对各种服务器。所以，本书的示例是文字界面的，它们应该在任何地方都能运行（其实仅限于Unix类的操作系统的命令行，如Linux。如果你使用其他操作系统，请自己找办法调出命令行）。

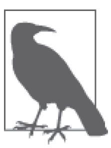
排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 等宽斜体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

使用代码示例

本书的所有程序和脚本都可从 http://mysqlresources.com/files 下载，并任由复制和修改。

本书旨在帮助你学习 MySQL 和 MariaDB，或完成 MySQL 和 MariaDB 的相关工作。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如：“*Learning MySQL and MariaDB* by Russell J.T. Dyer (O'Reilly). Copyright 2015 Russell J.T. Dyer, 978-1-449-36290-4.”

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920029175.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

感谢我的同事 Colin Charles、Kenneth Dyer、Chad Hudson、Caryn-Amy Rose 和 Sveta Smirnova。如果没有他们对草稿中技术等各种问题的审阅和建议，本书是不可能完成的。感谢我的编辑 Andy Oram，对我寄予厚望并在我们相识的多年里一直支持着我。感谢我的两位上司——曾经历过 MySQL Ab 和 SkySQL/MariaDB Ab 的 Ulf Sandberg 和 Max Mether，在他们的英明领导下，我工作得非常愉快。同时，还要感谢我的同事兼好友 Rusty Osborne Johnson，他在本书的创作上也耐心地给予我不少帮助。

电子书

扫描如下二维码，即可购买本书电子版。

第一部分

软件

MySQL 和 MariaDB 最主要的部分是服务器。在这里，*服务器*指的是软件，而不是指运行这个软件的计算机。该服务器可以维护、控制和保护你的数据，将它们以各种格式存储在文件中，而这些文件正是放在运行服务器的计算机上。服务器监听来自其他软件的请求（这种情况下，这些软件被称为客户端）。在这里，*客户端*指的是软件，而不是计算机。客户端和服务器可以在同一台计算机上同时运行，该计算机可以是一台笔记本电脑。

刚开始，我们会使用命令行客户端，在其中手工输入请求。接着，我们会从其他程序中发起请求，这些程序包括备份软件以及其他操作数据的软件。你不需要通晓构成 MySQL 的所有文件和程序。不过，还是有一些关键部分是需要注意的。

其中一个关键程序，就是服务器 `mysqld`（d 代表 daemon，即守护进程，通常服务器都是守护进程）。MySQL 和 MariaDB 的守护进程都叫 `mysqld`。该守护进程必须一直运行着，这样人们才能访问并修改数据。如果你是管理员，你就有能力进行配置，使 `mysqld` 符合你建立数据库系统的需求。关于守护进程的介绍穿插于本书的多个相关章节中。

而另一个关键程序，就是基本的 MySQL 客户端，简称 `mysql`，本书中经常用到它。有了它，你就可以与 `mysqld`，即数据库，进行交互。它的界面是文本式的。它朴实无华，连鼠标都不用。基本上你就是用它来敲打在本书中学到的 SQL 语句。语句的结果会以 ASCII 形式展示，看上去非常简洁，不掺杂任何图形。同时，因为只有文字（没有二进制和图像文件），所以它运行得很快。我们会在第 3 章讲到它。当然 GUI 客户端也是有的，但大多数 MySQL 开发者和管理员都倾向于 `mysql`，而且用 GUI 客户端发送的命令其实跟你在 `mysql` 里输入的没有区别，所以我就不介绍它了。

第 1 章
入门

MySQL 是一个开源、多线程、关系型数据库管理系统，由 Michael“Monty”Widenius 于 1995 年开发出来。2000 年，MySQL 以双重许可形式发布，根据 GNU 的通用公共许可（GPL），人们可以免费使用。再加之功能繁多、运行稳定，MySQL 迅速普及。

据估计，全球 MySQL 的装机量超过 600 万，另据报告，MySQL 每天的下载量超过 5 万。作为一个领先的数据库，MySQL 的成功不仅因为它的价格——毕竟，免费的开源数据库不只有它——还因为它稳定可靠、性能高、功能强大。不过，MariaDB 很快就要取代 MySQL 了，很多人都觉得它将继承 MySQL 社区的精神。

如果你的工作是计算机编程、网页开发或者与更广义的计算机技术有关，那么学习 MySQL 和 MariaDB 将对你有益。很多企业在开发和维护定制软件时都会用到 MySQL。除此之外，不少流行的网站和软件也用 MySQL，或者是某种与 MySQL 概念相通的数据库。所以，无论你是数据库开发者还是网页开发者，都很可能需要或受益于 MySQL 方面的知识。因此，学习 MySQL 和 MariaDB 是你计算机职业生涯的重要基础。

1.1 MySQL和MariaDB的价值

MySQL 有很多非凡的特性，尤其是速度（可参考 http://www.mysql.com/why-mysql/benchmarks/ 中各时期的基准测试）。MySQL 和 MariaDB 扩展性极高，可以处理上万个表和上亿行数据。当数据量不多时，它们同样快速流畅，非常适合小型业务或业余项目。

任何一个数据库管理系统的关键部分都是存储引擎，因为它要管理所有查询，以及用户 SQL 语句与数据库后端存储之间的接口。MySQL 和 MariaDB 提供了几种各有优点的存储引擎。其中有些带有事务功能，允许回滚数据（就像桌面软件中我们熟悉的“撤销”操作）。MySQL 还内置了大量函数，这些我们会在本书的不同章节中详述。而 MariaDB 的内

置函数则在 MySQL 的基础上还多出一些。另外，MySQL 和 MariaDB 快速稳定的更新也是出了名的，每一版的更新都会给大家带来新功能，以及速度和稳定性的提升。

1.2 邮件列表和论坛

学习 MySQL 和 MariaDB 时，特别是你在工作中初用 MySQL 时，懂得上哪儿寻求协助是很重要的。遇到数据库的问题，你可以到 Oracle 建立的一些 MySQL 论坛社区（http://forums.mysql.com/）获得免费帮助。你首先应该进行注册，以便发问或帮助别人。帮助别人会使你学到很多东西，因为这会加深你对 MySQL 的了解。同样，类似的资源还有 MariaDB Ab（https://mariadb.com/resources/community-tools）。

有疑问时，你可以在论坛上查找别人是否问过类似的问题。在发问之前，最好先搜索论坛和文档。如果实在找不到，再提问。你还要注意将问题发在相关的版块里。

1.3 其他书籍和出版物

MariaDB 提供的在线文档（https://mariadb.com/kb/en/mariadb/documentation/），一般也适用于 MySQL。Oracle 自己所有的产品都有详尽的在线文档（http://dev.mysql.com/doc），包括 MySQL 服务器。这些文档因 MySQL 版本不同而异。你可以在线阅读，也可以按各种格式下载（如 HTML、PDF、EPUB）。对于 PDF 和 EPUB，你可以将其下载到电子阅读器中。我也维护着一个网站（http://mysqlresources.com/），里面包含一些文档，以及衍生自我另一本书《MySQL 核心技术手册》的一些示例。另外也有一些人在这里补充了其他示例和资料。

除了本书，O'Reilly 还出版了其他一些值得入手的 MySQL 相关书籍。它主推的 MySQL 参考书是我写的《MySQL 核心技术手册》。如果需要解决现实中的常见问题，可参考 Paul DuBois 的《MySQL Cookbook（中文版）》。若是需要调优和性能监控方面的建议，如数据库备份，则可参考 Baron Schwartz、Peter Zaitsev 和 Vadim Tkachenko 合著的《高性能 MySQL》。我曾与这两本书的作者在 MySQL 公司共事，他们都是 MySQL 方面的权威，并且在圈中备受敬仰。

O'Reilly 也出版过几本 MySQL 应用编程接口（API）的相关书籍。对于 PHP 和 MySQL 开发，有 Robin Nixon 的 *Learning PHP, MySQL, JavaScript, CSS, and HTML5*（2014，http://shop.oreilly.com/product/0636920036463.do）。关于 Perl 与 MySQL 及其他数据库的接口，则有 Alligator Descartes 和 Tim Bunce 的《Perl DBI 编程》（这本书的原版在 2000 年出版，但现在依然有价值）。想用 Java 连接 MySQL，可以用 JDBC 和 JConnector 驱动；而 George Reese 的《JDBC 与 Java 数据库编程》是这方面一个不错的参考。

除了书籍之外，一些网站也提供了 MySQL 的简明教程。顺便说下，我也在 O’Reilly 博客和其他相关出版物上发表过几篇有关 MySQL 的文章。MySQL 的网站亦提供了一些深入介绍（http://dev.mysql.com/tech-resources/articles），其中不少都是讲新产品和新特性的，甚至还包括处于测试阶段的东西。如果你想了解最新版本，这些文章会很合适你。除了投入时间阅读，所有这些在线出版物都不要求你支付任何费用。如果你是 MySQL 的支持客户，

你还可以访问它的私有智库（我也在其中做了多年的编辑）。

掌握本书的内容之后，如果你还想获得一些进阶的培训，MariaDB Ab 提供了培训课程。MariaDB Ab 的培训页面（http://www.skysql.com/products/mysql-training）列出了世界各地的课程，以及它们的开始时间，其中有些是一到两天的课程，有些是持续一周的。顺便说下，我是 MariaDB Ab 现任的课程管理人。

第 2 章

安装MySQL和MariaDB

MySQL 和 MariaDB 的服务器与客户端，能在好几种操作系统上运行，准确来说，是某些 Linux 的发行版、Mac OS X、FreeBSD、Sun Solaris 和 Windows。

本章会简单地介绍如何在 Linux、Mac OS X 和 Windows 上安装 MySQL 和 MariaDB。关于如何在某些操作系统的各种发行版安装，本章有专门的小节讲述。在任何一种操作系统上，安装 MySQL 都只需阅读这三节内容：2.4 节、2.5 节和 2.6 节。没有必要了解所有版本的安装方法。

2.1 安装包

MySQL 和 MariaDB 的安装包带有几个程序。其中最主要的是服务器程序，即 mysqld 守护进程[1]。它在 MySQL 和 MariaDB 中是同名的。这个守护进程是对整个数据库进行存储和操控的实际执行者。它监听着一个特定的端口（默认是 3306），这个端口供用户提交查询。标准的 MySQL 客户端就叫作 mysql。用户可使用它的命令行接口登录和执行 SQL 查询。该客户端还能接受含有查询命令的文本文件，代表用户或其他软件执行查询。不过，我们大多使用各种编程语言来跟 MySQL 交互。有关 Perl、PHP 以及其他语言的 MySQL 接口，会在第 16 章中讲到。

在服务器安装目录中，会有一些封装好的脚本。而运行 mysqld 最常见的方法就是使用其中的 mysqld_safe 脚本，因为它能自动重启崩溃的守护进程。这有助于令数据库服务的宕机时间最小化。如果你是初学者，那么你无需掌握其中的运作细节，但告诉你这些，是让你知道这套数据库系统是多么强大。

注 1：所谓守护进程，就是一个在后台持续运行的进程，这是 Unix 中的术语，而大多数人就叫它“服务器”。

MySQL 和 MariaDB 都自带各种服务器管理工具。`mysqlaccess` 用于创建用户账号和设置权限。`mysqladmin` 则是命令行的数据库服务器管理工具。你可以用它来交互式地查询服务器的状态和使用量，以及关闭服务器。 而 `mysqlshow` 既可显示各数据库和各表的信息，又可查看服务器状态。其中有些工具需要先在服务器上安装 Perl；如果用 Windows，则要安装 ActivePerl。你可到 Perl 的网站（http://www.perl.org/）或 ActivePerl 的网站（http://www.activestate.com/activeperl）上下载安装。

另外，还有一些导入导出数据的工具。`mysqldump` 是最流行的导出 dump 文件（包含表结构和数据的纯文本文件）的工具。这使得我们可以备份数据，或在服务器之间复制数据。而 `mysql` 客户端则可以将 dump 文件导回到数据库中。具体内容在第一部分中有详解。

助手工具是可以不装的。但它们不是什么大型文件，安装它们也不耗费什么，所以你也可以安装并使用。

2.2　许可

MySQL 是免费并且开源的，但它的开发者——现在是 Oracle 公司——对其源代码拥有版权。该公司提供双重许可方式：一种是可在某些常见情境下遵循 GPL 的免费使用，另一种是收费的商业许可证。同一软件，却有两种不同的许可和权限。有关 GPL 的细节，可在其创造者自由软件基金会的网站上找到（http://www.fsf.org/licenses/license-list.html）。

如果你仅仅使用 MySQL 而没有重新发布它，又或者是重新发布时所包含的软件都遵循 GPL，那么这都是符合 GPL 的，Oracle 允许你这么做。你甚至可以在重新发布 MySQL 时带上你自己开发的软件，只要它也遵循 GPL 即可。正因如此，MariaDB 才得以诞生，并被认可为 MySQL 的合法分支。

然而，如果你开发了一个依赖 MySQL 的应用，并希望以它来盈利，那么你必须向 Oracle 购买一个商业许可。除此之外，还有其他一些情形是需要商业许可的。详情请浏览 MySQL 法律网页（http://www.mysql.com/about/legal/）。

除了版权，Oracle 还拥有 MySQL 的商标。因此，你不能发布名字含有 MySQL 的软件。这些对于学习使用 MySQL 并不重要，但对于高级 MySQL 开发者来说是需要注意的。

2.3　获取软件

你可从 MySQL 网站（http://dev.mysql.com/downloads/mysql/）或镜像站（http://dev.mysql.com/downloads/mirrors.html）获取该软件，但你得先有个 Oracle 账号（免费的）。你也可以下载 MariaDB，它包括 MySQL 的最新功能以及其他一些特性。它在 MariaDB 基金会网站（https://downloads.mariadb.org/mariadb/）提供下载，同样，它也是免费的，也需要先注册。

这两个网站在下载的过程中，都会要求你提供个人信息、组织背景，以及把该软件作何用途。这些信息会被收集到它们的营销部门。但你也可以表明你不想被打扰，仅仅下载完就好，不用再与他们打交道。

如果你的服务器或本地计算机上已装有 MySQL 或 MariaDB，那么你可以跳过本章。如果

你不确定有没有装，在 Linux 或 Mac 上，可通过命令行这样检查：

```
ps aux | grep mysql
```

如果 MySQL 在运行着，你会看到类似下面的结果：

```
2763 ?        00:00:00 mysqld_safe
2900 ?        5-23:48:51 mysqld
```

在 Windows 中，你可用 `tasklist` 检查。只需在命令行键入：

```
tasklist /fi "IMAGENAME eq mysqld"
```

如果 MySQL 在运行着，你会看到类似下面的结果：

```
Image Name          PID  Session Name     Session#   Mem Usage
==============  =======  ==============  =========  ==========
mysqld.exe         1356  Services                0       212 K
```

如果没有运行，则会看到这样的信息：

```
INFO:  No tasks are running which match the specified criteria.
```

但这并不完全保证你没有安装 MySQL，它只是告诉你 MySQL 的守护进程没在运行。你可用文件管理器之类的工具在计算机中搜索 `mysqld`。你还可以使用 `mysqladmin`（假设你安装了）来检测 MySQL。键入下文第一行，你会看到接下来的结果（示例而已）：

```
mysqladmin -p version status

mysqladmin  Ver 9.0 Distrib 5.5.33a-MariaDB, for Linux on i686
Copyright (c) 2000, 2013, Oracle, Monty Program Ab and others.

Server version          5.5.33a-MariaDB
Protocol version        10
Connection              Localhost via UNIX socket
UNIX socket             /var/lib/mysql/mysql.sock
Uptime:                 30 days 23 hours 37 min 12 sec

Threads: 4  Questions: 24085079  Slow queries: 0  Opens: 10832  Flush tables: 3
Open tables: 400  Queries per second avg: 8.996 Uptime: 2677032  Threads: 4
Questions: 24085079  Slow queries: 0  Opens: 10832  Flush tables: 3
Open tables: 400  Queries per second avg: 8.996
```

如果你从以上测试得知 MySQL 已运行，那么你可以跳到第 3 章了。否则，你可能需要启动它。具体做法会在 2.5 节中各小节末尾处介绍。你需找到符合你的 MySQL 版本的小节（如 Mac OS X），然后翻到小节末尾，看如何启动守护进程。然后试着启动它。如果启动成功，那就跳到本章末尾的 2.6 节。那里有一些重点关注，包括安全问题。如果启动不了，那么就要通读符合你版本的整节了。

2.4 挑选发行版

在下载之前，请先想好要安装哪个版本的 MySQL（或 MariaDB）。对于 MySQL 来说，最好是按照 Oracle 的建议，下载最新的稳定版，即正式版（GA）。这对于新手来说是最佳选

择。初学者不要碰测试版或开发版。除非你有 Oracle 的支持（即有 MySQL 企业版），否则应使用 MySQL 社区版。初学者用社区版和企业版其实是没什么区别的。

对于 MariaDB 来说，也要下载正式版（https://downloads.mariadb.org/mariadb/）。

可以选择用源码安装或用二进制执行文件安装。我们推荐用二进制文件安装，因为这比较简单。如果要在编译或安装过程中做一些特别设置，又或者是没有适合你的操作系统的二进制包，则要选择用源码安装。否则，就用二进制文件安装吧。当前我们只为学习 MySQL 的基础知识，没必要在安装上花太多工夫。

2.5 各种_AMP

接下来的小节会介绍在不同操作系统中，以不同的方法和形式下载和安装 MySQL（或 MariaDB）。不过，有个简单的方法，就是使用 _AMP 包。这个缩写代表 Apache、MySQL/MariaDB 和 PHP/Perl/Python。其中 Apache 是最流行的 Web 服务器，而 PHP 是与 MySQL 适配的最流行的编程语言。AMP 包（栈）基于操作系统：Linux 的叫 LAMP，Mac 的叫 MAMP，Windows 的叫 WAMP。安装 AMP 包时，除了 Apache、MySQL 和 PHP，还会顺带安装上相关依赖程序。这种做法很便捷，尽管你还是需要做一些安装后的调整设置（本章最后一节会介绍）。安装后，你就直接跳到 2.6 节吧。

刚才所说的包，可在下面这些网站上找到。

- 最新 Linux 版，请看 Apache XAMPP 网站（多出的 P 是指 Perl）：http://www.apachefriends.org/en/xampp-linux.html。尽管该网站将这个包称为 XAMPP 而非 LAMPP，但其实是一回事。
- 最新 Mac 版，请看 SourceForge MAMP 网站：http://sourceforge.net/projects/mamp/。
- 最新 Windows 版，请看 EasyPHP WAMP 网站：http://www.easyphp.org/download.php。

它们全都很容易安装，采用默认的设置通常没什么问题。

2.5.1 Linux二进制发行版

如果你的服务器的 Linux 通过 RPM（红帽软件包管理器）或 DEB（Debian Linux）来安装软件，那么我们建议你使用二进制包而非源码包来安装 MySQL。以下系统有特定的二进制包：RedHat、Debian 和 SuSE。而其他发行版则有通用的包。另外，还要区分服务器的处理器类型（例如 32 位或 64 位）。

不过，在继续下去之前，如果你有 Linux 原始的安装盘，那么你可以直接用它来安装 MySQL。这样安装完的话，你就可以跳到 2.6 节，而无需继续阅读本节后面的内容了。但若你的安装盘太老旧（里面不是最新版的 MySQL），你可能需要按照接下来介绍的方法去安装。

所有版本的 MySQL，都提供以下这些二进制安装包的下载：*MySQL 服务器*、*共享组件*、*兼容库*、*客户端工具*、*嵌入式*，以及*测试套件*。其中最重要的是 MySQL 服务器、客户端工具、共享组件。此外，你可能还想安装*共享库*。若想使用 PHP、Perl、C 之类的编程语

言来与 MySQL 进行交互，那么这个共享库就是必要的。其他包服务于那些高级或特殊的需求，本书不作介绍，等你学完基本的内容再说吧。

这些包的命名规则一般是：MySQL-server-`version`.rpm，MySQL-client-`version`.rpm，MySQL-shared-version.rpm。其中 `version` 是实际的版本号。供 Debian 安装的包名则是以 .deb 结尾，而非 .rpm。

安装包下载到服务器上之后，就可用 `rpm` 来安装了，又或者用更强大的工具 `yum` 来安装。`yum` 之所以更好，是因为它能避免你安装与现有软件冲突的东西，自动帮你安装上缺失的依赖，还能检查新版本并升级。对 Debian 系统来说，类似的工具有 `apt-get`。Oracle 已为 MySQL 建立了 yum 库（http://dev.mysql.com/downloads/repo/yum/）和 apt 库（http://dev.mysql.com/downloads/repo/apt/）。而 MariaDB 则有对应各系统的库配置工具（https://downloads.mariadb.org/mariadb/repositories/）。

键入类似以下的命令，用 `yum` 来安装 MySQL 二进制包：

```
yum install MySQL-server-version.rpm \
MySQL-client-version.rpm MySQL-shared-version.rpm
```

当然，文件名是可以改的。`yum` 会在一路上跟你确认各种安装、移除、冲突和升级问题。除非有特殊要求，否则你可以按默认的设置进行安装。

键入类似以下的命令，用 `yum` 来安装 MariaDB 二进制包：

```
yum install MariaDB-server MariaDB-client
```

在含有 RPM 包的目录里，键入类似以下的命令，用 `rpm` 来安装 MySQL 或 MariaDB 二进制包：

```
rpm -ivh MySQL-server-version.rpm \
MySQL-client-version.rpm MySQL-shared-version.rpm
```

如果服务器上已安装有一个旧版 MySQL，那么安装过程就会报错并中止。如果想升级，你可用 `-U` 选项（大写）来替换 `-i`：

```
rpm -Uvh MySQL-server-version.rpm
MySQL-client-version.rpm MySQL-shared-version.rpm
```

RPM 文件安装好后，`mysqld` 守护进程就会自动启动或重启。MySQL 安装好并运行后，你还需要做一些安装后的调整（在 2.6 节中有详解）。所以，现在就可以过去看看了。

2.5.2 Mac OS X发行版

以前 Mac OS X 是自带 MySQL 的，但最近的版本没有（从 Oracle 接管 MySQL 开始）。如果你的计算机运行的是旧版本，那么就可能已经安装好了 MySQL，但没运行起来。要想检查有没有安装 MySQL，打开终端（在应用 / 工具中）。出现命令提示符后，键入下文第一行（第二行至第四行是结果）：

```
whereis mysql mysqld mysqld_safe

/usr/bin/mysql
```

```
/usr/bin/mysqld
/usr/bin/mysqld_safe
```

若结果如上，那就是安装了。接着检查 MySQL 守护进程（`mysqld`）有没有运行，键入：

```
ps aux | grep mysql
```

如果 `mysqld` 在运行，那就没必要再装了，请跳到 2.6 节。

如果没在运行，输入以下命令，以 root 身份启动它：

```
/usr/bin/mysqld_safe &
```

如果你的 Mac 上没装过 MySQL 或者想升级，那么接着看本节。对于没安装的人来说，你需要在装之前先创建一个叫 mysql 的用户。而 Oracle 的 MySQL 包会自动创建 _mysql 用户。

我们可通过 Mac 的 DMG 文件来安装 MySQL。对于没有 GUI 或桌面管理器的 Mac 服务器，或需要远程操作的情况，可用 TAR 文件来安装[2]。不管你是下载 DMG 还是 TAR 文件，都应注意处理器类型要求（如 32 位或 64 位），以及操作系统的版本要求（如 Mac OS X 10.6 或以上）。

如果装有旧版 MySQL，你就需要在安装、运行新版 MySQL，或替换成 MariaDB 之前，先将其关掉。为此，你可以使用 MySQL 管理器，它是一个 GUI 程序，可能会跟随 MySQL 一起安装。最近的 Mac OS X 一般都有它。如果你没有 MySQL 管理器，则需要键入以下命令来关掉 MySQL：

```
/usr/sbin/mysqladmin -u root -p shutdown
```

如果你从没用过 MySQL，也没设置过密码，那么密码应该是空的。当你键入上述命令后，被要求输入密码时，直接敲 Enter 键即可。

要想安装 MySQL 包，可从 Finder 双击你下载的镜像文件（DMG），这会打开里面的内容。然后查找 PKG 文件，正常来说会有两个。双击名为 mysql-*version*.pkg 的那个（如 mysql-5.5.29-osx10.6-x86.pkg）。这就开始安装了。对大多数用户和开发者来说，安装过程一切按默认设置即可。

若要 MySQL 开机时启动，可加上*启动项*。在你下载的 DMG 中，有个 MySQLStartupItem.pkg。双击它，即可为 MySQL 建立一个启动项。你还应该安装一个 MySQL 偏好窗格，以便能从 Mac 的系统偏好中启动和关闭 MySQL，或设置其是否开机自启。为此你需要点击 MySQL.prefPane。如果安装有问题，可参考 DMG 里的 ReadMe.txt。

Mac 现在还没有 MariaDB 的官方安装工具。不过你可以用 `homebrew`（http://brew.sh/）来下载和安装所需的包（其中包括所需的库）。Mac 的 `homebrew` 如同 Linux 的 `yum`。装好 `homebrew` 后，你就能使用以下命令来安装 MariaDB 了：

```
brew install mariadb
```

注 2：`tar` 是出自 Unix 的归档工具，但很多系统上的很多归档工具都能识别这种文件格式。

想用 TAR 而非 DMG 来安装的话，先从 Oracle 网站下载 TAR，然后将其移到 /usr/local 目录，并进入该目录。接着，这样解包：

```
cd /usr/local
tar xvfz mysql-version.tar.gz
```

上述文件名请按实际情况填写。然后，为该安装目录创建一个软连接，并执行设置程序。以下是例子：

```
ln -s /usr/local/mysql-version /usr/local/mysql
cd /usr/local/mysql

./configure --prefix=/usr/local/mysql \
  --with-unix-socket-path=/usr/local/mysql/mysql_socket \
  --with-mysqld-user=mysql
```

第一行命令创建软连接是不带版本号的，而源目录是有版本号 `version` 的（请按实际情况填写）。为源目录加上版本号，比较方便日后查找。当然你也可以不加版本号，但这样很难管理新旧版本的目录。

第二行命令会使你跳转到安装目录。第三行命令运行配置程序安装 MySQL。这里我已经加入了一些有助于解决疑难的选项。按个人需要，你也可以加上更多选项，但对于大多数新手来说，这些就够了。

然后，你还要设置这些文件和目录的所有者，以及组权限。用户和组都设为 `mysql`，它们应该在安装时就被建好了。对于某些系统，可能需要先用 `vsdbutil` 来开放硬盘和卷的权限。如果想先检查一下，可用 `-c` 选项。如果想直接开放，用 `-a` 选项。另外你还需要在 /usr/bin 中建立 `mysql` 和 `mysqladmin` 的软连接：

```
vsdbutil -a /Volumes/Macintosh\ HD/

sudo chown -R _mysql /usr/local/mysql/.

alias mysql=/usr/local/mysql/bin/mysql
alias mysqladmin=/usr/local/mysql/bin/mysqladmin
```

上例第一行用于开放 Mac 的主驱动器的权限。当然，你安装 MySQL 的位置可能不叫 Macintosh HD。第二行用于改变所有者。最后两行用于创建别名，以便你能从任何目录运行它们。

现在，你应该已经可以启动守护进程，并登录进 MySQL（或 MariaDB）了。如果你安装了 MySQL 偏好窗格，还可以通过系统偏好来启动。

```
sudo /usr/bin/mysqld_safe &
mysql -u root -p
```

对于不同版本的 MySQL，安装 dmg 的路径（上例第一行）可能会不一样。`&` 符号用于将进程放在后台运行。第二行开启 `mysql` 客户端并让你以 root 身份登录，这个 root 是 MySQL 的最高级管理员，与操作系统的 root 无关。该命令要求你输入密码，但密码应该是空的，所以直接按 Enter 键即可登录。

成功的话，就说明你已经正确设置了 `mysql` 软连接，并能连接 MySQL（或 MariaDB）。在正式使用之前，我们还有一些事情要做。所以，先输入 `exit` 并按 Enter 键，退出 `mysql` 吧。

现在 MySQ（或 MariaDB）已然装好，请参阅 2.6 节，做安装后的调整。

2.5.3　Windows发行版

在 Microsoft Windows 服务器上安装 MySQL 或 MariaDB 相当简单。你可从 MySQL 网站找到各种安装方式和版本，下载的安装包包含所有相关组件。MariaDB 基金会网站也有提供用于 Windows 服务器的安装包。最简单的做法就是下载并使用 MySQL Installer for Windows。它会帮你搞定一切。旧版本以 TAR 文件方式提供，而 Installer 的方式更方便，而且是最新版。无论是 TAR 还是 Installer，都分 32 位和 64 位，你需要按自己服务器处理器的类型来选择。

Installer 和 TAR 都含有 MySQL（或 MariaDB）的必要文件，包括本书提及的所有命令行工具（如 `mysql`、`mysqladmin`、`mysqlbackup`），适用于特殊任务的便利脚本，以及一些 API 库。它们还为你提供了对应版本的 /usr/local/mysql/docs。

如果你打算用 TAR 来安装，就要在开始时额外做一些人工操作，因为它不像 Installer 那么智能。首先，解压缩该 TAR 文件，以得到里面的安装文件。这一步需要你先在机器上安装有 WinZip（http://www.winzip.com）之类的解压工具。然后，把安装文件复制到 c:\mysql。如果没有这个目录，那么你还需要自己去建立。再使用文本文件编辑器（如 Notepad），在 c:\windows 中建立配置文件，一般名为 my.ini。配置文件的样例在 TAR 中可找到。这些文件都放好后，就可以启动 setup 了。虽然它也很自动，但还是不如 Installer。

如果你的服务器上已经有 MySQL 并且在运行了，而你想安装一个新版本，那么在启动 Installer 或 setup 之前，你得先关掉原本那个。在 server 版的 Windows 上，MySQL 通常就是一个服务。你可以在命令窗口中输入如下命令来关闭并移除它：

```
mysqld -remove
```

如果它不是作为服务而运行着，那就用以下命令来关掉它：

```
msyqladmin -u root -p shutdown
```

如果报错的话，你可能需要搞清楚 `mysqladmin` 的绝对路径，然后改成这样：

```
"C:\Program Data\MySQL\MySQL Server 5.1\bin\mysqladmin" -u root -p shutdown
```

下载好 Installer 之后，只需双击该图标即可启动。而对于 ZIP，则是双击解压的 setup.exe。它们接下来的运作基本一样。

开始安装并确认许可问题之后，它会让你选择安装类型。一般推荐*开发者*选项。但它并不包含 API 和一些工具。它只会安装 MySQL 服务器、库以及几个客户端。通常这是最佳选择。虽然 MySQL 体积也不大，但如果你平时是从其他桌面远程操作这台服务器的话，那么你可以选择“只安装服务器”，使其在服务器上安装 MySQL 服务器。然后在其他桌面选择“只安装客户端”。你也可以在服务器上选“只安装服务器”，另外在桌面安装开发包。

这样能方便你在桌面上完成开发测试后，才将应用发布到服务器。选择搭配请结合自身实际情况。最起码要安装好 MySQL 服务器和客户端，以使你能连上数据库。

在选择安装类型的同时，你还要选两个文件路径：一个放工具，另一个放数据文件。你可以按默认，也可以自定义到其他驱动器或位置。通常默认设置就很好。记得把路径复制出来，因为你以后可能需要翻查。虽然在配置文件中也有，但现在方便就现在复制吧，省得你以后再找。

接着，Installer 会检查你的机器上有没有缺少必要的文件。不管它要装什么，都让它装就是了。而 TAR 方式的文件路径选择，一般是 C:\Program Data\MySQL\ 放程序，C:\Program Data\MySQL\MySQL Server *version*\data\ 放数据，其中 *version* 是你实际的版本号。

Installer 结束之前，还会出现一个设置界面。如果需要，你可以点击“高级设置”，但因为你是初学，所以最好还是不点，让它按默认的设置来。其实设置是可以日后改动的。

如果你的安装含有 MySQL 服务器，你会看到“开机启动 MySQL”的勾选框。我们建议打勾。而在配置那一步中，你可以为 root 设置密码。设置密码要谨慎，并牢牢记住。你还可以在此添加其他用户。添加用户会在 2.6 节中介绍。但如果你想简单一点，也可以在这一步里添加——不过我还是建议用安装后的 MySQL 来添加，因为那才是常用技能。剩下的选项，采用默认设置就行了。

本书使用命令行来教学，所以你要时时都能方便地找到命令行。为了能在任何目录中开启命令行工具，输入以下语句：

```
PATH=%PATH%;C:\Program Data\MySQL\MySQL Server version\bin
export PATH
```

把 `version` 替换成实际的版本号，并注意路径要准确。如果改过默认路径，请用改后的。上例使你输入 `mysql` 即可打开客户端，而不用每次都写一大串的 C:\Program Data\MySQL\MySQL Server *version*\bin\mysql。注意某些 Windows 版本的路径是以 C:\Program Files\ 开头的。可执行文件的具体路径可搜索 bin\。已经打开的命令行窗口是不会获得这个新路径的，你需要开个新的窗口来检查看看。

Installer 会在完成安装并建立好配置文件后，自动开启 MySQL。手动安装的人，要这样来启动：

```
mysqld --install
net start mysql
```

现在 MySQL 已经装好并且在运行了，请跳到本章 2.6 节，做一些后期调整。

2.5.4 FreeBSD和Sun Solaris发行版

MySQL 或 MariaDB 的二进制包比源码包更易安装。如果你的平台有相应的二进制安装包，那请使用它。对于 Sun Solaris 来说，Oracle 网站提供了 MySQL 的 PKG 文件，MariaDB 基金会网站提供了 MariaDB 的 PKG 文件。MySQL 需要根据你的处理器区分 32 位、64 位和 SPARC。MariaDB 则只有 64 位版本。

TAR 包也是有的。FreeBSD 方面则只有 MySQL 的 TAR；想安装 MariaDB，你需要自己编译源码。下载的 TAR 需要用 GNU 的 `tar` 和 `gunzip` 来解压出安装文件。这两个工具通常已置于 Sun Solaris 和 FreeBSD 中。如果没有，你可从 GNU 基金会网站（http://www.gnu.org）下载。

挑好并下载好安装文件后，以 root 身份运行以下命令来开始安装：

```
groupadd mysql
useradd -g mysql mysql
cd /usr/local
tar xvfz /tmp/mysql-version.tar.gz
```

上述命令同时适用于 MySQL 和 MariaDB。第一行创建名为 mysql 的用户组。第二行创建用户 mysql，并将其加到 mysql 组中。第三行转到用于解压 MySQL 的目录。最后一行用 `tar`（加上 `-z` 以使用 `gunzip`）来解压和提取安装文件。`version` 处请填实际版本号，也就是说，使用你下载得到的文件的名字和实际的文件路径，作为 `tar` 的第二个参数。在 Sun Solaris 上，请用 `gtar` 替换 `tar`。

执行上述命令之后，接着为创建出的目录建立软连接：

```
ln -s /usr/local/mysql-version /usr/local/mysql
```

这会使 /usr/local/mysql 指向 /usr/local/mysql-`version`，其中 mysql-`version` 是刚刚 `tar` 释放出的目录。这么做是有必要的，因为 MySQL 会默认从 /usr/local/mysql 找执行文件，而从 /usr/local/mysql/data 找数据文件。

现在，MySQL（或 MariaDB）已经基本上安装好了。接下来要生成初始的用户权限或对表进行授权，以及改变相关程序和数据文件的所有权：

```
cd /usr/local/mysql
./scripts/mysql_install_db

chown -R mysql /usr/local/mysql
chgrp -R mysql /usr/local/mysql
```

第一行命令先跳到 MySQL 所在的目录。第二行使用发行版自带的脚本，生成初始的用户权限或对表进行授权，包括建立 MySQL 的超级用户——root。这在 MariaDB 中也是一样的。第三行将 MySQL 目录和项目的所有者改为 mysql。最后一行将该目录的组改为 mysql。

安装好程序并设置好权限后，就可以启动 MySQL 了。这有好几种做法。为保证守护进程能在崩溃后自动重启，使用下面这个命令：

```
/usr/local/mysql/bin/mysqld_safe &
```

此命令所启动的 `mysqld_safe` 守护进程，会发起 MySQL 服务器守护进程——`mysqld`。一旦 `mysqld` 崩溃，`mysqld_safe` 就会重启它。结尾的 `&` 符号能使整条命令在后台运行。这样你登出服务器后，MySQL 也依然会继续运行。

想要 MysQL（或 MariaDB）开机启动，可将 /usr/local/mysql/support-files 的 mysql.server 文件复制到 /etc/init.d 中。你会用到以下命令：

```
cp support-files/mysql.server /etc/init.d/mysql
chmod +x /etc/init.d/mysql
chkconfig --add mysql
```

第一行是惯例做法，将启动文件放到服务器的守护进程初始化目录，并改名为 mysql。第二行使该文件可执行。第三行将其启动级别定为开机和关机。

这样 MySQL（或 MariaDB）就安装好并且能运行了。接着请到 2.6 节看看怎样做一些后期调整。

2.5.5 源码包

尽管我们推荐使用二进制包来安装 MySQL 和 MariaDB，但有时你还是会想用源码包，也许是因为没有适合你的操作系统的二进制包，又或者是你有特别需求而要修改安装文件。在类 Unix 系统（包括 Linux、FreeBSD 和 Sun Solaris）上安装源码包的步骤基本上是一样的。本小节会讲解这些步骤。

在安装源码包之前，你得先准备好 GNU 的 gunzip、tar、gcc（v2.95.2 或以上）和 make。Linux 系统和大多数 Unix 系统一般都带有这些工具。如果没有，请从 GNU 基金会网站（http://www.gnu.org）下载。

挑选并下载好源码包后，在你想要释出源码的目录，以 root 执行以下命令：

```
groupadd mysql
useradd -g mysql mysql
tar xvfz /tmp/mysql-version.tar.gz
cd mysql-version
```

以上命令也适用于安装 MariaDB，只是名字需要改成 mariadb-5.5.35.tar.gz 之类，从 TAR 释出的目录名也会有所不同。第一行创建组 mysql。第二行创建用户 mysql，同时将其加到 mysql 组。下一条命令使用 tar（加上 -z 以使用 gunzip）来解压和提取源码。version 请填实际版本号。第二个参数的文件路径和名字也请按实际情况填写。最后一条命令跳转到刚刚解压出的目录。该目录包含了配置 MySQL 的必要文件。

下一步，设置源码文件以便构建出二进制文件。这一步里你可以根据特殊需求来加入自定义的一些东西。比如说，你想改变默认安装目录，可以使用 --prefix 选项来指定。想改变 Unix 套接字文件的路径，可以用 --with-unix-socket-path 指定。如果你不想使用默认的 latin1 字符集，可以用 --with-charset 另作指定。以下是上述需求的示例：

```
./configure --prefix=/usr/local/mysql \
            --with-unix-socket-path=/tmp \
            --with-charset=latin2
```

你可以一行打完而不用反斜线。除此之外，还有其他配置选项。想获知全部，可用此命令：

```
./configure --help
```

又或者参考“编译 MySQL”的网上文档（http://dev.mysql.com/doc/refman/5.6/en/compilation-problems.html）。

决定好配置什么后，就可以用该脚本来执行了。它运行起来会花一些时间，并打印出一大堆信息，如果最后成功结束的话，那么这些信息是可以不用看的。结束之后，我们开始构建二进制文件，以及初始化 MySQL：

```
make
make install
cd /usr/local/mysql
./scripts/mysql_install_db
```

第一行构建二进制文件。它跟第二行都会输出大量信息，这里为了节省空间就不印出来了。成功后，执行第二行，以安装二进制文件和相关目录里的文件。第三行跳到安装目录。如果你改过安装目录，那请填新目录。最后一行使用安装好的脚本，来做用户和表的权限初始化设置。

剩下要做的就是更改程序和目录的权限：

```
chown -R mysql /usr/local/mysql
chgrp -R mysql /usr/local/mysql
```

第一行将 MySQL 程序和目录的所有者改为 mysql。第二行将组改为 mysql。其中的文件路径会因 MySQL 版本不同而异，如果你改变过配置，那么也会不一样。

都完成之后，就可以启动守护进程了。这里做法也有多种。为保证守护进程能在崩溃后自动重启，使用以下命令：

```
/usr/local/mysql/bin/mysqld_safe &
```

此命令对 MySQL 和 MariaDB 都适用，它启动的 `mysqld_safe` 守护进程会发起服务器守护进程 `mysqld`。一旦 `mysqld` 崩溃，`mysqld_safe` 就会重启它。结尾的 `&` 符号能使整条命令在后台运行。这样你登出服务器后，MySQL 也依然会继续运行。

想要 MysQL 或 MariaDB 开机启动，可将 /usr/local/mysql/support-files 的 mysql.server 文件复制到 /etc/init.d 中。你会用到以下命令：

```
cp support-files/mysql.server /etc/init.d/mysql
chmod +x /etc/init.d/mysql
chkconfig --add mysql
```

第一行是惯例做法，将启动文件放到服务器的守护进程初始化目录，并改名为 mysql。第二行使该文件可执行。第三行将其启动级别定为开机和关机。

这样 MySQL（或 MariaDB）就安装好并且能运行了。接着请到下一节看看怎样做一些后期调整。

2.6 安装后

安装好 MySQL 或 MariaDB 后，在让别人使用之前，还有一些事情要做。例如，可能你会想修改配置。或者至少，你会想改变 root 的密码，还要创建一些非管理员的用户。某些版本的 MySQL 会预先提供一些匿名用户，那些你应该删掉。本节将会介绍这些任务。

尽管 MySQL 和 MariaDB 的作者已经给出了推荐的配置，但你或许会做一些个性化设置。例如，开启错误日志功能。

2.6.1　特殊配置

想要开启错误日志之类的功能，你需要修改 MySQL 的主配置文件。在类 Unix 系统中，它是 /etc/my.cnf。在 Windows 中，则是 C:\windows\my.ini 或者 C:\my.cnf。主配置文件应该用文本文件编辑器来修改——切勿用文字处理软件，那样会引入隐含的二进制字符，这将导致一些问题。

配置文件被分成多个小节（小组），每节前面都有一个用方括号括起来的标题。例如服务器守护进程的设置在 `[mysqld]` 之下。在此标题下，你可以加入类似 `log=/var/log/mysql` 的东西，以设置日志文件的路径，并激活日志功能。一节里可以有很多配置项。以下是一个样板：

```
[mysqld]
datadir=/data/mysql
user=mysql
default-character-set=utf8
log-bin=/data/mysql/logs/binary_log
max_allowed_packet=512M

[mysqld_safe]
ulimit -d 256000
ledir=/usr/sbin
mysqld=mysqld
log-error=/var/log/mysqld.log
pid-file=/data/mysql/mysqld.pid

[mysql.client]
default-character-set=utf8
```

新手可能不需要改动配置。暂时你只需要知道这个配置文件的存在与所在，以及如何修改它。最重要的是修改 root 的密码，因为它起初是空的。

2.6.2　给root设置初始密码

改变 root 密码的方法有多种。其中一种是使用管理工具 `mysqladmin`：

```
mysqladmin -u root -p flush-privileges password "new_pwd"
```

`new_pwd` 处请替换成一个强密码。如果你得到类似“mysqladmin command is not found”的提示，可能是因为你没有为 MySQL 的目录创建软连接，或者是该目录不在 PATH 中。那么请参照前面相应发行版里介绍的做法来修正。或者你可使用完整路径。在 Linux 或类 Unix 系统上，试着运行 `/usr/local/mysql/bin/mysqladmin`。在 Windows 上，试试 `c:\mysql\bin\mysqladmin`。

不过，如果你是在一台联网的机器上操作，最好还是别用这个命令来修改，因为可能有人通过偷看或者翻查服务器日志来窃取密码。对于 v5.5.3 的 MySQL，你可以这么做：

```
mysqladmin -u root -p flush-privileges password
```

执行这条命令之后，它会提示你输入旧密码，因为初始是空的，所以直接按 Enter 键。接着它会两次提示你输入新密码。这种方法不会将你的密码显示在屏幕上。如果一切都安装好了并且 `mysqld` 在运行着，那么输完以后应该是不会出现任何信息的。

MySQL 的 root 跟操作系统的 root 是完全不同的两个概念（虽然名字一样）。它只在 MySQL 和 MariaDB 中有意义。在本书所有地方，当我说 root 时，一般就是指 MySQL 的 root。当要说操作系统的 root 时，我会特别指出。

2.6.3 关于密码的更多问题，以及删除匿名用户

MySQL 中的权限是根据用户名和主机来决定的。例如，从 localhost 登录的 root 被允许做任何事，但远程登录的就做不了什么。这是为了安全着想。所以，root 的用户名与主机的组合就有好几种。因为你刚才是登录到服务器上，再使用 `mysqladmin` 来改密码的，所以密码对应的是 root/localhost。接着你要给剩下的组合设置密码。想查看所有组合，可用以下命令：

```
mysql -u root -p -e "SELECT User,Host FROM mysql.user;"

+------+-----------------------+
| User | Host                  |
+------+-----------------------+
| root | 127.0.0.1             |
| root | localhost             |
| root | %                     |
|      | localhost             |
+------+-----------------------+
```

如果运行不了，可能是你的 PATH 中没有 `mysql`。那就要在前面加上 `/bin/` 或者 `/usr/bin/`，或其他安装路径。MariaDB 也是用这个命令。这结果是我编出来的，你实际看到的会不一样。

很多版本的 MySQL 都含有 root 和 `%` 的组合，其中 `%` 是指任何主机。这就不安全了，因为它允许人们从各种地方通过 root 连接 MySQL。另外也有很多版本的 MySQL 含有空用户名加上 `localhost` 的组合，即任何用户名都可从 `localhost` 登录。这就是匿名用户。尽管用户已经存在，但初始密码都是空的。你要做的是删掉没用的用户，并给留下的设置密码。尽管 `127.0.0.1` 和 `localhost` 对应的是同一个主机，但你还是要分两次来改密码。对于上面的查询结果，若要给前两条组合改密码，并删掉后两条组合，你可在命令提示符执行以下命令：

```
mysql -u root -p -e "SET PASSWORD FOR 'root'@'127.0.0.1' PASSWORD('new_pwd');"
mysql -u root -p -e "SET PASSWORD FOR 'root'@'localhost' PASSWORD('new_pwd');"
mysql -u root -p -e "DROP USER 'root'@'%';"
mysql -u root -p -e "DROP USER ''@'localhost';"
```

当你修改完这第一批用户后，得用以下命令刷新一下，以使密码生效：

```
mysqladmin -u root -p flush-privileges
```

从此以后，root 就得用新密码来登录。

2.6.4 创建用户

接下来谈谈如何至少再建立一个用户作普通用途。我们最好不要用 root 来做数据库的日常管理。要建立另一个用户，用这条命令：

```
mysql -u root -p -e "GRANT USAGE ON *.*
TO 'russell'@'localhost'
IDENTIFIED BY 'Rover#My_1st_Dog&Not_Yours!';"
```

上例会建立名为 russell 的用户，并允许它从 localhost 登录 MySQL。其中 *.* 表示所有数据库和所有表。更详细的解释会在后面讲到。同时这里将其密码设置为 Rover#My_1st_Dog&Not_Yours!。

他现在还没有任何权限：不能查看数据库，更不用说写入数据。新建用户时，你需要考虑他能获得什么权限。如果你希望他只能查看数据，那么用这条命令：

```
mysql -u root -p -e "GRANT SELECT ON *.* TO 'russell'@'localhost';"
```

这使得 russell 只能使用 SELECT 语句（可查询数据）。如果想得知某个用户所拥有的权限，输入：

```
mysql -u root -p -e "SHOW GRANTS FOR 'russell'@'localhost' \G"

*************************** 1. row ***************************
Grants for russell@localhost:
GRANT SELECT ON *.* TO 'russell'@'localhost'
IDENTIFIED BY PASSWORD '*B1A8D5415ACE5AB4BBAC120EC1D17766B8EFF1A1'
```

结果表明，该用户只能使用 SELECT 语句来查询数据。我们会在后面更深入地讲解。注意这里显示的密码是加密后的。MySQL 永远都只返回加密后的密码。

上例的 russell@localhost，无法增加、修改或删除数据。如果你想给他查询以外的权限，可用逗号分隔来指定。这会在第 13 章谈及。现在为了赋予他所有权限，将 SELECT 改成 ALL，即：

```
mysql -u root -p -e "GRANT ALL ON *.* TO 'russell'@'localhost';"
```

这样，russell@localhost 就有了所有的基本权限。你应建立这样一个功能齐全的用户，以便在学习本书的过程中随便试验。不过请另外起个更衬你的名字。

经过一系列的下载、安装、配置和授权，MySQL（或 MariaDB）这个数据库系统已经能用了，接下来你就可以学习如何操作了。

第3章 基础知识与mysql客户端

我们可以通过各种不同的方法来跟 MySQL 或 MariaDB 数据库交互。MySQL 服务器的接口程序叫作 MySQL 客户端。这类客户端有很多，但本书专注于最适合交互式用户的那个——文本式 `mysql`。它是最常用的，高手都喜欢，一般也推荐新手使用。

还有带 GUI 的客户端，但终究没什么用。首先，使用 GUI 学不到什么东西。它们所给你的视觉提示或许能帮你完成一些基本的查询，但却无助于高级的工作。而文本式 `mysql` 则会驱使你更多地思考和记忆——而且这其实并不困难。更重要的是，GUI 经常改变外观。一旦“变脸”，你就要重新学习如何操作。如果你换工作了，或去到客户现场，或因为各种理由需要使用其他人的系统，你可能会发现新环境的 GUI 跟你惯用的不同。然而，`mysql` 客户端是无处不在的，因为它是与 MySQL 服务器一起安装的。所以，本书的示例都假设你用的是 `mysql`。我也建议当你看到示例时，试着用 `mysql` 把它们敲进计算机，以巩固学习。

3.1 mysql客户端

使用 `mysql` 客户端，你便能在命令行或监视器的环境下与 MySQL 或 MariaDB 交互。命令行方式不会损耗多少性能，而且还能执行脚本或其他程序。例如，你可以在 `cron` 中写命令，来执行一些维护性工作和自动备份数据库。监视器则是一个 ASCII 界面，它的文字经过格式化，而且它能提供你所执行的命令的详细信息。本书绝大部分示例都是从监视器截取出来的，剩下的那些我会特别注明它们来自命令行。

如果 MySQL 或 MariaDB 安装正确，那么 `mysql` 应该是可用的。如果不可用，请参考 2.6 节，确保一切都配置好，并建立必要的软连接或别名。`mysql` 应该放在 /bin/ 或 /usr/bin/ 目录里。对于 Windows、Mac 或其他有 GUI 的操作系统，可用文件定位工具来查找 `mysql` 和其他 MySQL 二进制文件。

即使 mysql 可用，你还需要一个 MySQL 用户名和密码才能连接 MySQL。如果你不是管理员，就需要找指定人员获取。如果 MySQL（或 MariaDB）刚刚才安装好，而且 root 的密码还没设置，那么该密码就是空的，你在提示输入密码时直接按 Enter 键就行。如需学习设置 root 密码和创建新用户并给他们授权，请参见 2.6 节，那里有一些基本的技巧，而第 13 章则有更详尽的细节。

3.2 连接到服务器

只要有了用户名和密码，你就可以用 mysql 连接 MySQL 了。举个例子，我给自己创建了一个叫 russell 的用户，那么我就可以在命令行上这样来连接：

```
mysql -u russell -p
```

理解这条命令的每一部分是有用的。-u 选项后面是你的用户名。注意它们之间用空格分开。这里你可以把 russell 替换成自己创建的用户名。我们所指的用户是 MySQL 用户，而不是操作系统用户。顺便提一下，使用 root 连接并不是一个好的安全实践，除非你要执行一些 root 才有权限执行的管理任务。所以如果你还没给自己建立 root 以外的用户，那么请先做这一步。登录 MariaDB 的命令也与 MySQL 的相同。

-p 选项会让 mysql 提示你输入密码。你可以在 -p 后面加上密码（如 -pRover#My_1st_Dog&Not_Yours!，其中 -p 后的就是密码）。这种做法要求 -p 和密码之间不留空格。然而，在命令行上输入密码也不是好的安全实践，因为这样密码就显示在屏幕上了（站在你身后的人就能看到了），而且这密码将在网络上以明文方式传输，同时，只要有人监控着服务器的进程列表，那么他也能看到。更好的做法是，-p 选项后不带密码，等 MySQL 询问时才输入。那么密码就不会显示在屏幕上，也不会被保存到其他地方了。

如果你要用的 MySQL 用户名与操作系统的相同，那么你不需要加上 -u 选项，只需 -p 就行。即是这样：

```
mysql -p
```

一旦输入了正确的 mysql 命令以及密码，你就能登录到 MySQL（或 MariaDB）。到时你会看到类似以下的内容：

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1419341
Server version: 5.5.29 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

如果你装的是 MariaDB，则是这样：

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 360511
Server version: 5.5.33a-MariaDB MariaDB Server, wsrep_23.7.6.rXXXX

Copyright (c) 2000, 2013, Oracle, Monty Program Ab and others.
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>>
```

在第一行里，“Welcome to the MySQL/MariaDB monitor”之后，说的是命令要以分号（;）或斜线 +g（\g）结尾。当你输入命令或 SQL 语句时，可以在任何地方按 Enter 键来开始新的一行，并接着输入剩余的命令。在遇到 ; 或 \g 前，mysql 是不会将你输入的东西发送到 MySQL 服务器的。如果你用的是 \G（大写 G），那就会是另一种格式，我们很快就会讲到。暂时我们只用分号。

第二行告诉你本次连接的标识号。如果出现问题的话，你可能会需要用到它。但现在我们先忽略它。

第三行告诉你 MySQL（或 MariaDB）的版本号。它能让你知道出问题时你应找哪个版本的网上文档。或者如果你在升级前想得知现行版本，那么也可以在这里找到。

下一行说的是获得在线帮助。在线帮助提供所有 SQL 语句和函数的帮助文档。你可试着输入它，看能输出些什么。

- `help`

  此命令会介绍如何使用 mysql。

- `help contents`

  此命令将 MySQL 或 MariaDB 的各种帮助分门别类地以列表展示出来。在列表中你会看到有个 Data Manipulation，里面就包含了有关增删改的 SQL 语句。

- `help Data Manipulation`

  此命令会将所有可用的数据操作语句显示出来，比如 SHOW DATABASES。

- `help SHOW DATABASES`

  此命令用于找到 SQL 语句相关的帮助。如你所见，mysql 提供了大量有用的信息。如果你忘了某个 SQL 语法，可以在这里快速地找到答案。

第三个 help 的结果里，讲到了一个简单但是有时却很好用的技巧：如果你想取消输入到一半的 SQL 语句，可接着输入 \c，然后无需加分号，直接按 Enter 键。这样就能清空 mysql 缓存着的那写到一半的命令，甚至分行输入的也能清空，使你回到 mysql> 提示符。

而最后一行的 mysql>，叫作*提示符*。它在提示你输入命令，学习本书的过程中你会一直与它打交道。如果你没有结束一条语句就按 Enter 键，提示符就会变成 ->，告诉你这条 SQL 还没发送给服务器。而在 MariaDB，默认的提示符则不是这样。它显示的是 MariaDB [(none)]>>。在你设定了默认数据库之后，其中的 none 自会变成当前默认数据库的名字。

顺便说一下，其实可以将提示符改成其他样子。为此，输入 prompt，它后面是你想要显示的文字。另外你还可输入一些特别标志（例如，\d 代表默认数据库）。下例可供参考：

```
prompt SQL Command \d>\_
```

于是提示符就会变成：

```
SQL Command (none)>
```

现在你还没有默认数据库。既然你已经启动了 mysql 客户端，那么我们就开始探索数据库吧。

3.3 开始探索数据库

下面几章涵盖如何新建数据库，并向其增加数据，以及从中查询一些有趣的关系。当你通过 mysql 登录 MySQL 或 MariaDB 后，就可从本章学习数据库系统的核心概念了。我们先关注几个基本概念，使你能在 mysql 监视器中输入一些命令。这样你会习惯 mysql 的用法。考虑到你不一定是高手，所以我们现在会尽量介绍简单内容。

先解释一个 SQL 术语——表，它是用于存储数据的，反映用户查看数据的方式。举个例子，在关于电影的表中，你会看到每一部电影都占一水平行，行中每列是电影的各种信息，并且每列都有一个标题。

```
+----------+---------------------+--------+
| movie_id | title               | rating |
+----------+---------------------+--------+
|        1 | Casablanca          | PG     |
|        2 | The Impostors       | R      |
|        3 | The Bourne Identity | PG-13  |
+----------+---------------------+--------+
```

这只用于举例，我们不用你建这样的表。现在来看看服务器上已经存在的东西。为此，我们在 mysql> 提示符处输入以下命令并按 Enter 键：

```
SHOW DATABASES;
```

接着就会出现这样（或类似这样）的回应：

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| test               |
+--------------------+
```

首先，我们先讲讲本书的约定。MySQL 不区分关键字（如 SHOW）的大小写，所以你可以用 show 甚至 sHoW。然而，数据库、表和列的名字却可能是区分大小写的，尤其是在那些大小写敏感的操作系统上，如 Mac OS X 或 Linux。大多数书籍和文档都会将关键字大写，但同时告诉你其实可以随意大小写。对于数据库、表和列的名字，我们全用小写，因为这样看着舒服，敲字也简单，更重要的是这样能令读者辨别出哪些是 SQL 约定的，而哪些是可变的。

刚刚显示的列表，告诉你现在一开始就有了三个数据库，它们是安装时自动创建的。information_schema 数据库包含服务器的相关信息。接着是 mysql 数据库，它则存储着用户名、密码和权限。你在第 2 章结尾给自己所创建的用户，其信息最终会放到这里。你也

许已注意到，第 2 章用到的一些命令会从这个数据库获取结果。但切勿直接修改该数据库。以后我会教你操作该库的命令。暂时我们只以管理函数和工具来访问它。最后那个叫 test 的数据库，用于做测试和练习。本章我们会轻度使用它。

3.3.1 第一条SQL语句

test 数据库一开始是空的，里面没有任何表。那么我们就来建一个。不用担心操作细节难不难懂，我会一步步讲解的。

好了，我们先在 mysql 输入以下命令（记得还有结尾的分号）：

```
CREATE TABLE test.books (book_id INT, title TEXT, status INT);
```

这就是你的第一条 SQL 语句。它在 test 数据库中新建一个名为 books 的表。其中 test.books 指定了数据库名和表名（例如，格式是*数据库．表*）。在括号里，我们还给这个表定义了三个列，以后会进行更深入的介绍。

如果输入无误，你会得到这样的回应：

```
Query OK, 0 rows affected (0.19 sec)
```

服务器返回的这个信息告知你，当 SQL 送出后，发生了什么。这里它的意思是一切顺利。既然顺利，我们就看看结果。想要列出 test 数据库所有的表，执行：

```
SHOW TABLES FROM test;
```

这会返回：

```
+----------------+
| Tables_in_test |
+----------------+
| books          |
+----------------+
1 row in set (0.01 sec)
```

现在你有了一个表——books。注意返回信息用了一些 ASCII 符号来作间隔，使其看上去像个表格，你可以照着它在纸上画出来。另外还请注意表格下的信息。它说该集合含有一行，意味着 books 是 test 中唯一的表。而服务器执行每条 SQL 语句所用的时间，都会显示在括号里。本例中我的服务器用了 0.01 秒。具体环境是，我用意大利米兰的家用计算机，连到美国佛罗里达州坦帕市的服务器。这算回得很快了。有时甚至更快，显示 0.00 秒，那是因为时间实在太短，无法精确到那种程度。

从现在开始，除非这些状态信息值得探讨，不然我会省略掉它们，以节约空间并使我们只关注重点。同样，我会连 mysql> 提示符也省掉。你需要分辨出哪些命令是从 mysql 客户端输入的，哪些是从操作系统的 shell 输入的——不过其实当要换成操作系统 shell 时，我还是会提示的。所以接下来，输入跟输出会是这样展示：

```
SHOW TABLES FROM test;

+----------------+
| Tables_in_test |
```

```
+----------------+
| books          |
+----------------+
```

其中粗体的是输入，非粗体的是输出。

在上面提到的 SQL 语句中，我们都必须指定数据库名。如果你基本上是在同一个数据库中操作（一般人都是这样），你可以设定默认数据库，这样就不用每次都输入数据库名了。具体做法是用 `USE` 命令：

```
USE test
```

> 顺便提一下，如果你的服务器没有 `test` 这个数据库，你可以先用 `CREATE DATABASE test;` 来建立。

因为这是 `mysql` 客户端的指令而非服务器的，所以我们通常不加分号结尾。客户端会告知服务器为它将默认数据库改成指定的那个。这样你就不必在指定表前指定数据库了——除非你想指定的是另一个数据库。执行完 `USE` 后，你可以重新输入刚刚的 SQL 语句，而不必指定 `test`。这是可以的：

```
SHOW TABLES;

+----------------+
| Tables_in_test |
+----------------+
| books          |
+----------------+
```

我们刚才已经看过了一个数据库，并创建了一个表，而数据库其实就是由一堆表来组成的（本例仅有一个表）。现在看看这个刚建的表。我们要使用到 SQL 语句 `DESCRIBE`：

```
DESCRIBE books;

+---------+---------+------+-----+---------+-------+
| Field   | Type    | Null | Key | Default | Extra |
+---------+---------+------+-----+---------+-------+
| book_id | int(11) | YES  |     | NULL    |       |
| title   | text    | YES  |     | NULL    |       |
| status  | int(11) | YES  |     | NULL    |       |
+---------+---------+------+-----+---------+-------+
```

在结果中，你会看到该表有三个域可供输入数据，它们分别名为 `book_id`、`title` 和 `status`。这看起来很有限，那是因为本章先从简单的讲起。其中第一和第三个域是整数类型，意思是它们只能存放数字。这是因为我们在建表时使用了 `INT` 关键字来规定。剩下的 `title` 则可以存放文本，包含了你能从键盘输入的任何东西。那是之前用 `TEXT` 关键字指定的。不用死记硬背，我们现在只是在这套系统中到处逛逛，玩玩 `mysql`，找找感觉。

3.3.2 插入和操作数据

让我们来放些数据进这个表。用 `mysql` 输入以下三个 SQL 语句：

```
INSERT INTO books VALUES(100, 'Heart of Darkness', 0);
INSERT INTO books VALUES(101, 'The Catcher of the Rye', 1);
INSERT INTO books VALUES(102, 'My Antonia', 0);
```

这三条语句都用到了 `INSERT` 来向 `books` 表中插入（或增加）数据。每一行都会返回一个状态信息（或者是错误信息，如果你输错命令的话），但我这里没把它们贴上来。注意数字是不需要用引号包围的，但文本却需要。这些语句的语法是非常结构化的，因此称为*结构化查询语言*。语句中各元素之间需要空格，数量随意，但顺序必须妥当，括号、逗号和分号也不能缺少，如上所示。因为使用结构化的语句能令查询容易识别，所以执行起来也更快。

上例会将括号里的值插入到表中。值的排列顺序和其格式，与建表时指定的一样：三个域，其中一、三是数字，二是任何文本。接着我们叫 MySQL 显示这些数据，看看会是什么样子：

```
SELECT * FROM books;

+---------+------------------------+--------+
| book_id | title                  | status |
+---------+------------------------+--------+
|     100 | Heart of Darkness      | 0      |
|     101 | The Catcher of the Rye | 1      |
|     102 | My Antonia             | 0      |
+---------+------------------------+--------+
```

从这个表格中，你可以更清楚地看到为什么它们被称为记录行和域列。我们使用了 `SELECT` 语句来从指定的表中选取所有列——星号（*）就是代表所有。本例中，我们拿 `book_id` 作为每条记录的唯一标识，而 `title` 所存的文本和 `status` 所存的数字才是我们真正关注的内容。`status` 的值为 0 或 1 是有目的的：0 表示闲置的，1 表示在用的。但其实设计是任意发挥的，MySQL 或 MariaDB 并不关心你的内容。顺便提一下，第二本书的标题是错的，但我们接下来会利用它来演示如何修改数据。

现在我们来试验一下 `SELECT` 和这些值，看看它们怎么运作。就加个 `WHERE` 子句吧：

```
SELECT * FROM books WHERE status = 1;

+---------+------------------------+--------+
| book_id | title                  | status |
+---------+------------------------+--------+
|     101 | The Catcher of the Rye | 1      |
+---------+------------------------+--------+
```

从结果可看出，我们只选取了 `status` 等于 1 的行（即选出在用的记录）。那就是 `WHERE` 子句的效果。这里 `WHERE` 属于 `SELECT` 语句的子句，而不能单独成句。接着试试这句，要求返回闲置的：

```
SELECT * FROM books WHERE status = 0 \G

*************************** 1. row ***************************
book_id: 100
  title: Heart of Darkness
 status: 0
*************************** 2. row ***************************
book_id: 102
  title: My Antonia
 status: 0
```

注意这次我们将 SQL 语句结尾的分号改成了 `\G`。我们在本章前面讲过，这也是一种结尾。它导致结果不以表格形式展示，而是使每条记录都分成多行来展示。有时候这样更易读，通常能避免因域的内容太长，令表格变得太宽而换行。不过这也只是偏好问题。

我们已经做到了给这个小小的表插入数据。现在请再来改一下数据。让我们修改某一行的 `status`。为此，需要用到 `UPDATE` 语句。它会产生两行状态信息：

```
UPDATE books SET status = 1 WHERE book_id = 102;

Query OK, 1 row affected (0.18 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

我们可以按语句的顺序来阅读和解析它们，这样能更好地读懂和记忆它们的语法。就拿上面代码块里的第一行语句为例。它的意思是，更新 `books`，将 `status` 的值设为 1，作用于 `book_id` 等于 102 的所有行。示例数据中只有一行是 102，所以接下来的信息告诉你，有一行受影响了，并且只有这一行被改变了，或者说被更新了——你可能更喜欢这种说法。为了看到结果，我们再运行一次之前的 `SELECT` 语句，即查看在用的那条语句：

```
SELECT * FROM books WHERE status = 1;

+---------+----------------------+--------+
| book_id | title                | status |
+---------+----------------------+--------+
|     101 | The Catcher of the Rye | 1      |
|     102 | My Antonia           | 1      |
+---------+----------------------+--------+
```

由于刚才的更新，我们现在有两行在用了。我们再执行一次 `UPDATE`，但使用另一个 `book_id`，将 *The Catcher in the Rye* 改成闲置：

```
UPDATE books SET status = 0 WHERE book_id = 101;

SELECT * FROM books WHERE status = 0;

+---------+----------------------+--------+
| book_id | title                | status |
+---------+----------------------+--------+
|     100 | Heart of Darkness    | 0      |
|     101 | The Catcher of the Rye | 0      |
+---------+----------------------+--------+
```

现在我们再做一次 `UPDATE`，让你看看怎样通过一条语句做更多的事。如前面说到的，有一

本书的标题错了。不应该是 *The Catcher of the Rye*，而应该是 *The Catcher in the Rye*。我们就来改改它在 `title` 内的文本，同时将它 `status` 的值改回 1。该效果可以通过两条语句实现，但让我们用一条搞定吧：

```
UPDATE books
SET title = 'The Catcher in the Rye', status = 1
WHERE book_id = 101;
```

注意这里的语法跟之前的 `UPDATE` 一样，但我们指定了要设置两对列和值。这样比输入两条语句更简单，同时也节省了与另一大陆的服务器沟通时的网络流量。

3.3.3 再复杂一点

让我们更进一步。建立另一个表，并插入两行数据。在 `mysql` 中，用这两个语句：

```
CREATE TABLE status_names (status_id INT, status_name CHAR(8));

INSERT INTO status_names VALUES(0, 'Inactive'), (1,'Active');
```

现在我们有了 `status_names` 表，但它只有两列。此 `CREATE TABLE` 语句与我们第一次建表时用的那个很相似。其中有一点不同是需要你注意的：这里没有用 `TEXT` 类型，而用了 `CHAR`，即是“字符”的意思。我们可以给这一列输入文本，但长度有限：每行的这一列都只允许最多八个字符。这使得该域更小，因此表也更小，处理起来也更快。不过这在我们的例子中没什么作用，因为我们没打算输入很多数据，但对于大型的数据库来说，这个细节却对性能有巨大的影响。你最好在设计的初始阶段就考虑好这种问题。

第二个语句为表增加了两组值。在单个 `INSERT` 语句中操作多组值是可以的，而且这样也比分开几行来写更简单。下面展示了该表数据的样子：

```
SELECT * FROM status_names;

+-----------+-------------+
| status_id | status_name |
+-----------+-------------+
|         0 | Inactive    |
|         1 | Active      |
+-----------+-------------+
```

这些数据看上去好像没什么用。但让我们来把它跟 `books` 表结合在一起，看看如 MariaDB 这样的数据库系统所隐藏的力量。我们将会用 `SELECT` 语句来连接两个表，以得出更好看的效果，并且此过程中我们有选择性地只让某些数据得以展示。试试在你的计算机上输入下面的内容：

```
SELECT book_id, title, status_name
FROM books JOIN status_names
WHERE status = status_id;

+---------+-----------------------+-------------+
| book_id | title                 | status_name |
+---------+-----------------------+-------------+
|     100 | Heart of Darkness     | Inactive    |
```

```
|     101 | The Catcher in the Rye | Active      |
|     102 | My Antonia             | Active      |
+---------+------------------------+-------------+
```

首先，注意我将这个 SQL 语句拆成了三行。这是可以的。在输入分号和按 Enter 键之前，没有任何语句会被执行。这样分行对人来说更加易读，而对 MySQL 是没什么影响的。在这个语句里，在第一行，我们选择了 book_id 和 title，它们都在 books 表，而 status_name 列则在 status_names 表。注意我们没有使用星号来选择所有列，而是指定了想要的列名。另外，这些列来自两个表。

在第二行，我们声明了这些列取自 books 和 status_name 表。其中 JOIN 子句用于指定第二个表。

在第三行的 WHERE 子句中，我们告诉 MySQL，将 books 表的 status 列的值，与 status_names 表的 status_id 列的值匹配起来。这使得两个表的行能连接起来。如果觉得这个概念难以理解，现在还不用担心。我只是为了演示 MySQL 和 MariaDB 的功能。关于连接，以后会做更详尽的解释。

在建立 books 表时，我们其实可以将 status 列声明为 TEXT 或 CHAR，并为每条记录打上 Active 或 Inactive。但如果你要为 books 输入成千上万行数据，显然打 0 或 1 更简单，而且不容易打错字（例如 Actve）。摆弄数据库是很无聊的，但如果你的表结构建得好，SQL 语句写得好，那么情况将有所改善，而且也能节省你的时间和资源。

3.4　小结

对于我们刚建出来的那些表，其实还有很多地方是可以探索的，但本章只是一次“闲逛”，只需你对 MySQL 和 MariaDB 有个大概的认识。从第二部分的第 4 章（其中有建表的细节介绍）开始，我们才会深入挖掘。

在往下看之前，你可能会想巩固刚刚学到的知识。下面有一些练习，你可以使用 mysql 客户端连接 test 数据库来做做。做完后，输入 quit 或 exit，再按 Enter 键，就可以退出 mysql 了。

3.5　习题

除了用 mysql 登录 MySQL（或 MariaDB）以及输入本章展示的 SQL 语句，这里还提供了一些习题，让你继续把玩 mysql 以及助你更好地理解那些基础知识。你要使用一些更符合实际的命名，而不是 books 和 book_id。为了贯彻这种精神，请在做练习时使用真实数据（如人名要叫“John Smith”）。

(1) 用 mysql 登录 MySQL 或 MariaDB，并将默认数据库切换到 test。建两个表，分别名为 contacts 和 relation_types。这两个表的数字列都用 INT 类型，字符列用 CHAR 类型。为 CHAR 指定一个你想要的最大长度，否则 MySQL 会使用 CHAR 的极限长度，但那并不实用。注意该长度必须足够容纳你接下来要输入的数据。如果你需要输入的内容既有数字又有字符（如电话号码中的连字符），那么使用 CHAR。contacts 表要有五个列：

name、phone_work、phone_mobile、email、relation_id。而 relation_types 表，则只有两列：relation_id 和 relationship。

建好这两个表后，用 DESCRIBE 语句看看它们的样子。

(2) 输入数据到刚才建的两个表中。首先是 relation_types 表。输入三行数据。第一列填成一位数的序号形式，第二列填文本：Family、Friend、Colleague。接着填 contacts 表，至少填五组虚构的姓名、电话和邮件地址，而在最后的 relation_id 列，填入能与 relation_types 的 relation_id 对应的一位数。并确认那三个 relation_id 都被用到。

(3) 执行两条 SELECT 语句，以获取那两个表的所有列的所有数据。然后再写一条 SELECT，要求只获取 contacts 中的姓名和邮件地址。

(4) 用 UPDATE 来修改刚才输入的数据。如果忘了怎样操作，可翻回本章前面看看。首先，修改某个人的姓名或电话。接着，修改某个人的邮件地址以及他（她）与你的关系（即 relation_id）。要求两次修改都分别只用一条 UPDATE 语句。

(5) 执行一条 SELECT 语句，连接习题 1 的那两个表。用 JOIN 子句来实现（本章讲到了 JOIN 子句，如果忘了请回去看看）。因为两个表都有 relation_id 列，所以就在这两列上连接——写在 WHERE 子句中。以下是提示：

```
...
FROM contacts JOIN relation_types
WHERE contacts.relation_id = relation_types.relation_id
...
```

只显示你的 Friend 的 name 和 phone_mobile 列——需要在 WHERE 中使用 AND。先试试按 relation_id 来筛选，再试试按 relationship 筛选。

第二部分

数据库结构

MySQL 和 MariaDB 的主要部分就是数据库。通常我们会为不同的企业、组织、部门或项目单独建立数据库。至于如何区分不同，全凭个人喜好。这样分别建库确实有利于为不同的用户（组）设置不同的访问权限。而对于新手来说，先学好为一个组织建一个数据库就够了。

之前在 3.3 节中提过，数据库可有多个*表*，每个表可有多*行*（或多*条记录*），每行可有多列（*域*），用于存放每个项目的各方面信息。与分别建库不同，分别建表是一种惯用的策略。有些新手喜欢在一个数据库中建立一个拥有很多列的大型表，但其实这样处理数据是很低效的。现实中，仅建立一个表是不合理的。所以，还请建立多个小表，而非一个大表（这里的大是指列很多）。

在建表时，我们需要规划好*方案*，即需要建什么域（或列）。声明列时，是有很多选项的。最起码，你要指定其类型：包含字符还是只包含整数？包含日期和时间，还是二进制数据？同时，你还能指定怎样为该列建立索引，是按字母序（拉丁文或是中文），还是其他因素？

本部分的第 1 章（即第 4 章），讲述了如何建数据库——这很简单——以及如何建表。另外我还会讲解怎样把数据放进表中，以及怎样从表中取出数据。这些话题会延伸至后面很多章节。如果只教你如何建表，而不教如何用表，那过程会非常枯燥。所以要在深入到细节之前，先告诉你为什么要建表。

刚开始建表时，通常很难确定方案，对于新手来说尤其是这样。所以我们总是会在建好后又做修改。因此，在第 5 章，我们会看看如何在建表后改表。虽然我可以将改表的章节放在数据操作的章节之后再讲，但这样的话，当你发现表建得不对时，就要在本书中跳来跳去了。

第 4 章

创建数据库和表

在插入和操作数据之前，你得先建好一个数据库。这没什么好讲的，只是建立一个存放表的容器。建表才复杂，你需要做出各种选择。表有好几种类型，其中有些功能独特。在建表时，你需要决定其结构：列的数量、每列的数据类型、索引形式，以及其他因素。不过，因为你还处于学习阶段，对于大多数的选项，你都可以采用默认设置。

以下是创建时需要考虑的一些基本问题。

- 数据库中要建多少个表，以及需要多少个表名。
- 每个表中要建多少列，以及需要多少个列名。
- 每一列存储的数据类型。

对于最后那个问题，因为现在只是刚开始，所以我们只会接触四种类型：数字、少量字符（不超过 255 个）、大量文字或二进制文件，以及日期和时间信息。对于创建数据库和表来说，这是一个很好的起点。随着学习的深入，我们会认识到更多的数据类型，并使用它们来提高数据库的性能。

本章含有建库和建表的示例。我们需要你使用 `mysql` 输入那些 SQL 语句。而本章结尾的习题，则要求你修改你计算机上的数据库和其中的表。所以，当你看到示例和习题时，都请上机试试。

本章建立的数据库和表，会在后面很多章节中用到，尤其是第三部分。到时候，我会让你对这些表的数据进行存取和修改。章后习题也会使用本章这些表。因此，为了发挥本书的最大价值，还请你按部就班地完成每一章的练习。这将使你读得更深刻，学得更全面。

4.1　创建数据库

创建一个数据库是很简单的，因为它真没什么复杂的东西。你只要用到 SQL 语句 `CREATE`

DATABASE，再加上一个名字即可。名字可以很随意，例如 db1。然而，还是做得真实一点、有趣一点吧。例如，我是一个鸟类爱好者，所以我假设我要做一个供观鸟网站使用的数据库。有些鸟是群居的，有鸟巢的。于是，我建一个叫 rookery 的数据库，以存放各种鸟种信息。为此，先从 mysql 客户端输入以下命令：

```
CREATE DATABASE rookery;
```

如之前所述，这句短小、初阶的 SQL 语句会在 MySQL 的 data 目录中建立一个 rookery 子目录。这不会产生任何数据，而只是划出一块放数据的地方。顺便说一下，如果你不喜欢 DATABASE 这个关键字，那么可以用 SCHEMA：CREATE SCHEMA database_name。它们的结果是一样的。

虽然我说很简单，但你也可以把它复杂化。例如，加多一些选项，来指定默认的字符集，以及指定数据的排序或校对方式。好吧，我们删掉 rookery 数据库，再重建一个：

```
DROP DATABASE rookery;

CREATE DATABASE rookery
CHARACTER SET latin1
COLLATE latin1_bin;
```

第一行的 CREATE 与之前的无异——但注意，它是与后两行合为一体的，因为到了第三行结尾才有分号出现。而第二行我们第一次见，它会告诉 MySQL 本库的表所默认使用的字符是拉丁文及其他字符。第三行则告诉 MySQL 数据的存储方式是二进制拉丁字符。关于二进制字符和二进制排序，我们会在后面章节讲到，现在还没必要掌握。事实上，最简短的那种写法已能满足大多数需求。后面所指定的那两个选项，是可以在建库后修改的。我现在提出，只是想让你知道这东西存在并可以由你自定义。

既然我们建好了一个数据库，那么就去确认一下它是否真在 MySQL 服务器上。想要查看数据库列表，可用以下 SQL 语句：

```
SHOW DATABASES;

+--------------------+
| Database           |
+--------------------+
| information_schema |
| rookery            |
| mysql              |
| test               |
+--------------------+
```

结果表明，我们有了 rookery 和另外三个安装 MySQL 时自建的数据库。那另外三个，我们在 3.3 节中已经介绍过了，如果以后有需要，会再提到。

在给 rookery 建表前，先从 mysql 输入以下命令：

```
USE rookery
```

它会将刚刚建好的数据库设为 mysql 的默认数据库，直至你再做更改，或退出客户端。这使得我们更方便地用 SQL 进行创建表或其他与表相关的操作。如果不这么做的话，我们就

要在每条语句中都指定数据库。

4.2 创建表

组建数据库的下一步就是建表。它可以很复杂，也可以很简单，我们先从简单的开始学起。开始时，我们会建一个用表，以及两个用于参考的小表。主表会有很多列，参考表则只有几个列。

我们的观鸟网站，其关注点就是鸟。所以，我们要建一个存放鸟种基本信息的表。因为只是一个习作，所以这个表我们不会做得很详尽。在 mysql 输入以下命令：

```
CREATE TABLE birds (
bird_id INT AUTO_INCREMENT PRIMARY KEY,
scientific_name VARCHAR(255) UNIQUE,
common_name VARCHAR(50),
family_id INT,
description TEXT);
```

此 SQL 语句会建立一个含有五个域（列）的表，其中每列的信息以逗号分隔开来。注意所有列的定义都被放在同一对括号中。每一列，我们都指定了名称和类型，某些列还带上了可选设定。例如，对于第一列，我们做了以下指定。

- 名称：bird_id
- 类型：INT（整数）
- 设定：AUTO_INCREMENT 和 PRIMARY KEY

名称可以是 SQL 保留字以外的任何东西。事实上，用保留字也可以，但需要加上引号以作区分。可选的数据类型，可在 MySQL 和 MariaDB 的网站上找到，我的书《MySQL 核心技术手册》里也有介绍。

我们只为这个表建了五列。实际上你可以定更多（最多 255 列），但不建议这么做。太多列的表，用起来麻烦，访问速度也慢。最好还是拆成多个表。

birds 表的第一列是一个简单的标识号 bird_id。通过 PRIMARY KEY 关键字，我们将它作为主键，使数据能以其索引。关于主键的重要性，我们会在后面讨论。

AUTO_INCREMENT 选项则告诉 MySQL 此列的值是自增的。如果没指定一个起始数，那么就是从 1 开始。

下一列存放每种鸟的学名（Charadrius vociferus 就是一个学名，而 Killdeer 则不是）。你可能会觉得我们不需要 bird_id，而应该用 scientific_name 作为索引。然而，学名可能会很长，包含拉丁字母或希腊字母（说不定两种都有），但不是人人都懂这两种语言。如果要人输入学名来查找记录，那就难搞了。另外，我们还将学名列设为变长字符类型（VARCHAR）。而 VARCHAR 后的括号里的 255 则指定了最大长度（255 应该够用了）。

如果我们要保存的学名小于 255 个字符，存储引擎为该行的该列分配的空间就会自动减少。它与 CHAR 不同。CHAR 分配到的空间总是最大长度，不会因为内容少而减少。使用哪一种类型，是需要权衡的。

用 CHAR 的话，存储引擎就能确切知道该列会返回多少东西，这会令表运行得更快，索引也更容易维护。而 VARCHAR 对硬盘空间的需求更少，利用效率更高。这也是提高性能的一种手段。如果你能确定某列内容的长度，那就用 CHAR，否则用 VARCHAR。

接着，common_name 列被指定为变长字符类型，并且长度最多为 50 个字符。

第四列的 family_id 用于存放鸟科的标识号，以标记该鸟种属于哪一科。该列使用整数类型（INT）。之后我们会再建一个表，用于记录科的更多相关信息。然后我们就可以在操作数据时，用科的标识号对两个表进行连接，得知每种鸟所属的科。

最后一列是每种鸟的描述，用 TEXT 类型，即长度可变，但最多 65 535 字节。这样我们就能为每种鸟输入大量的描述，写上好几页都没问题。

当我们想在数据库中找某一种鸟时，可能会想到很多因素，所以我们可以再给该表加更多的列：关于迁徙模式的信息，关于如何在野外识别它们的信息，等等。除此之外，数据类型的选择也有很多。我们可以指定某列能存放极大的数，或只能存放不大的数，或指定它能存放二进制文件。例如，二进制类型使得我们可以为每种鸟都存放一张照片。现在建得这么简单，是为了让你以后建表时能从这里扩展。

想看该表建成什么样，用 DESCRIBE。它会展示一个表各列的信息（或者说表结构）——不是指表的数据内容。为了查看我们刚建的那个表的结构，用以下语句：

```
DESCRIBE birds;

+-----------------+--------------+------+-----+---------+----------------+
| Field           | Type         | Null | Key | Default | Extra          |
+-----------------+--------------+------+-----+---------+----------------+
| bird_id         | int(11)      | NO   | PRI | NULL    | auto_increment |
| scientific_name | varchar(255) | YES  | UNI | NULL    |                |
| common_name     | varchar(50)  | YES  |     | NULL    |                |
| family_id       | int(11)      | YES  |     | NULL    |                |
| description     | text         | YES  |     | NULL    |                |
+-----------------+--------------+------+-----+---------+----------------+
```

注意结果展示在以 ASCII 字符组成的表格之中。它看起来并不炫，但洁净、快速，而且信息完整。现在我们不看内容，而看其布局。

第一行是各列的标题。第一列，Field，用于展示表的所有列名。

第二列，Type，用于展示各列的类型。注意我们之前在括号里定义的长度，在这里也有显示（如 varchar(255)）。而 INT 的长度是我们没有指定的，这里就显示了默认值。INT(11) 的意义以及其他可选值，会在以后介绍。

第三列，Null，用于说明各列能否含有 NULL 值。NULL 意味着什么都没有，不存在数据。它与空白或空内容不同。这听起来很怪异。现在先记着它们是不同的，以后我们就会在实践中看到。

第四列，Key，用于说明该列是否是键——索引列。如果此处为空，则对应的列不是索引列，如 common_name。如果是索引列，则会显示具体的索引类型。因为空间有限，所以单词被截短显示。本例中 bird_id 是主键，所以截短成 PRI。而 scientific_name 则被设为另

一种索引，UNIQUE，于是截短显示为 UNI。

倒数第二列，Default，用于说明各列的默认值。默认值可以是各种值的集合。刚才建 birds 表时，我们没有指定默认值。默认值的设定可以在建表时做，也可以在建表后做。

最后一列，Extra，用于提供一些额外的信息。在本例里，我们看到它说 bird_id 是自增的。通常我们看到的也就是这样。

如果对现有的表结构不满意，可以使用 ALTER TABLE 语句（第 5 章会介绍）来修改它。如果你做错了，想推倒重来，可以将整个表删掉再重建。想要完全删掉一个表（包括数据），可以用 DROP TABLE 语句，并指定一个表名。请谨慎使用该语句，因为它将导致该表的数据完全丢失，并且这是不可逆转的。

> 顺便提一下，使用 mysql 时，你可以按向上的方向键来获取之前输入的每一行命令。所以说，当你建了一个表，但 DESCRIBE 发现错误，并将其删掉后，你可以用向上键来找回建表的语句，然后用向左键，将光标移到错误的地方并做一些修复。修改完后按 Enter 键，改过的 CREATE TABLE 就会被发送到服务器。

4.3 插入数据

上一节有很多东西需要你消化。所以，现在先歇一下，学学有关插入数据的问题。我们会用到 INSERT 语句，第 3 章做过简短的介绍，下一节会详细介绍。暂时，不要担心 INSERT 的所有写法。你只需用 mysql 将以下语句输入服务器：

```
INSERT INTO birds (scientific_name, common_name)
VALUES ('Charadrius vociferus', 'Killdeer'),
('Gavia immer', 'Great Northern Loon'),
('Aix sponsa', 'Wood Duck'),
('Chordeiles minor', 'Common Nighthawk'),
('Sitta carolinensis', ' White-breasted Nuthatch'),
('Apteryx mantelli', 'North Island Brown Kiwi');
```

这会为六种鸟建立六行数据。接着用下面的语句，看看表的内容。

```
SELECT * FROM birds;

+---------+----------------------+-------------------+-----------+-------------+
| bird_id | scientific_name      | common_name       | family_id | description |
+---------+----------------------+-------------------+-----------+-------------+
|       1 | Charadrius vociferus | Killdeer          |      NULL | NULL        |
|       2 | Gavia immer          | Great Northern... |      NULL | NULL        |
|       3 | Aix sponsa           | Wood Duck         |      NULL | NULL        |
|       4 | Chordeiles minor     | Common Nighthawk  |      NULL | NULL        |
|       5 | Sitta carolinensis   | White-breasted... |      NULL | NULL        |
|       6 | Apteryx mantelli     | North Island...   |      NULL | NULL        |
+---------+----------------------+-------------------+-----------+-------------+
```

如查询结果所示，MySQL 按我们的要求在两列中填充了数据，而在其他列中填了默认值

（NULL）。这些值我们都可以以后修改。

接着，在另一个数据库中再建一个表。我们已经有了 rookery 库的 birds 表来保存鸟种的信息，那就再建一个观鸟爱好者的数据库吧。这个数据库就叫 birdwatchers，里面包含一个叫 humans 的表，这样看起来就跟 birds 对应了：

```
CREATE DATABASE birdwatchers;

CREATE TABLE birdwatchers.humans
(human_id INT AUTO_INCREMENT PRIMARY KEY,
formal_title VARCHAR(25),
name_first VARCHAR(25),
name_last VARCHAR(25),
email_address VARCHAR(255));
```

这个表不算很详细，没有收集太多关于会员的信息，先暂时这样是够用的。接着就是输入数据。用以下语句录入四个会员：

```
INSERT INTO birdwatchers.humans
(formal_title, name_first, name_last, email_address)
VALUES
('Mr.', 'Russell', 'Dyer', 'russell@mysqlresources.com'),
('Mr.', 'Richard', 'Stringer', 'richard@mysqlresources.com'),
('Ms.', 'Rusty', 'Osborne', 'rusty@mysqlresources.com'),
('Ms.', 'Lexi', 'Hollar', 'alexandra@mysqlresources.com');
```

这就输入了四个人了。注意我们把第一列留 NULL，这样 MySQL 就能自动地为每行分配一个递增标识号了。

我们已建好了一些简单的表，虽然可以搞得更复杂，但这样就足够了，而且可以更好地理解表及其结构。

4.4　更深入地理解表

除了使用 DESCRIBE，我们还有另一种方法查看表结构。你可以使用 SHOW CREATE TABLE 语句。它基本上就是重新展示你（可能是在另一个数据库中）CREATE TABLE 的命令。其中比较有趣及有用的地方是，它还告诉你服务器采用了什么默认设置，如果你建表时有些选项没指定的话。以下是 SHOW CREATE TABLE 语句的使用方法及结果：

```
SHOW CREATE TABLE birds \G

*************************** 1. row ***************************
       Table: birds
Create Table: CREATE TABLE `birds` (
  `bird_id` int(11) NOT NULL AUTO_INCREMENT,
  `scientific_name` varchar(255) COLLATE latin1_bin DEFAULT NULL,
  `common_name` varchar(50) COLLATE latin1_bin DEFAULT NULL,
  `family_id` int(11) DEFAULT NULL,
  `description` text COLLATE latin1_bin,
  PRIMARY KEY (`bird_id`),
  UNIQUE KEY `scientific_name` (`scientific_name`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 COLLATE=latin1_bin
```

如前面所述，每一列都是可以指定很多选项的。如果你不指定的话，服务器就会采取默认选项。从这里你就可以看出这些默认选项。注意我们没有为任何一列指定默认值（除了第一列自增），于是默认值就被设为 NULL 了。而第三列，其字符集（字母、数字及其他字符）就被设为了 `latin1_bin` 格式。另外两列都采用了默认设置。这来源于我们在本章开头建库时（第二条 `CREATE DATABASE`）的设定。我们可以将列的字符集指定成与数据库的不同，但这通常没有必要。

你可能会发现，结果中 `bird_id` 列的那些选项里没有说明它是主键，但我们建表时确实指定了。其实，在这些列的说明的后面，就有键和索引的说明。这里显示了它有主键索引，并且是基于 `bird_id` 的。接着我们看到唯一键。对于这种键，它的索引的名称会跟该列一样（括号中是列名）。索引可基于多列来建，但现在我们只用一列。关于索引，我们会在第 5 章讲述。

`SHOW CREATE TABLE` 的结果还有一点需要我们注意。在最后一行的右括号后，有一些其他设定。首先，是表类型，或者说是该表使用的存储引擎的类型。这里我们用到的是 MyISAM，它是多数服务器的默认选项。你的服务器所默认的可能不是这个。不同的存储引擎对数据的存储和处理方式是不一样的。它们各有优缺点。

剩下两个是表的默认字符集（`latin1`）和默认校对方式（`latin1_bin`）。这直接或间接地来源于建库的默认值。你可以为一个表设定不同于库的字符集或校对方式，甚至是为某一列设定。

下面说明一下显式指定字符集和校对方式的好处。假设你的数据库服务于英国的观鸟爱好者群体，这样那些鸟的俗名就基本上是用英文来写了。但它后来却发展到吸引了欧洲其他国家的爱好者，于是你可能会想俗名需要容纳其他语言。比方说你想为土耳其的网站建个表，那么你就需要不同于英文的字符集和校对方式，因为土耳其语中除了拉丁字母，还有其他字母。

这时，字符集你可以选 `latin5`，它含有拉丁字母和其他字母。而校对方式，则可选 `latin5_turkish_ci`，它会根据土耳其语的字母表来排序。为了避免增加列时忘了指定字符集和校对方式，你可以对整个表设置 `CHARSET` 和 `COLLATE`。

在进行下一步之前，我还想再说说关于 `SHOW CREATE TABLE` 的一点：如果你想要创建一个与默认设定很不一样的表，可以拿 `SHOW CREATE TABLE` 的结果来作为起点，去构造一个更详尽的 `CREATE TABLE` 语句。它还经常被人用来查看服务器建表时到底在背后帮我们做了什么，是基于哪些安装时设定。

我们接着要建的表是 `bird_families`。它用于存放鸟的科信息。它能与 `birds` 表的 `family_id` 列进行关联。这样我们就能避免在 `birds` 表的每一行都录入科名和科的其他相关信息。

```
CREATE TABLE bird_families (
family_id INT AUTO_INCREMENT PRIMARY KEY,
scientific_name VARCHAR(255) UNIQUE,
brief_description VARCHAR(255) );
```

这个表建有三列。第一列是我们最关注的。它是索引列，而且会被 `birds` 表参考。这听起来好像是说它与 `birds` 表在物理上有连接，但其实不是。连接只会发生在我们执行 SQL 语

句查询两个表的一刻。在该 SQL 语句中，我们会基于两个表的 family_id 来做连接。这样，我们就能获得鸟种的列表，以及每种鸟对应的科，或者获得某一科的鸟种列表。

现在我们可以将一个科的所有信息放到一行中。而在为 birds 表输入数据时，只需在其 bird_id 列放置一个用于参考 bird_families 表的标识号。这样有利于数据一致性：因为填字母比填数字更易打错字。而且，birds_families 中一行的数据能被 birds 中几百行数据所复用，这样也节省了空间。很快我们就会看到这是如何做到的。

scientific_name 列用于存放该科的学名（如 Charadriidae）。第三列则是存放俗名（如 Plovers）。但人们对于某一科通常也有很多叫法，就像对某一种鸟有很多叫法一样。所以这一列就名为 brief_description。

接着再建一个关于鸟的目表。它是对科的分类。表名是 bird_orders。我们可以在其上面试试之前提到的一些额外选项。输入以下 SQL 语句：

```
CREATE TABLE bird_orders (
  order_id INT AUTO_INCREMENT PRIMARY KEY,
  scientific_name VARCHAR(255) UNIQUE,
  brief_description VARCHAR(255),
  order_image BLOB
) DEFAULT CHARSET=utf8 COLLATE=utf8_general_ci;
```

暂时先建四列吧。第一列，order_id，是被 bird_families 表参考的键。接着是 scientific_name，存放该目的学名，用 VARCHAR 类型，同时指定了最大长度。该长度超出了我们的实际需要，但因为我们估计不了描述到底需要多长，而且该表应该不会有太多条目，所以就用最大的吧。名称叫作 brief_description，跟 bird_families 表一样。

因为至今建立的表也都有着类似的列名（如 scientific_name），所以可能会在连接时导致一些问题。我们似乎可以通过起不同的名字来解决这个问题（如 order_scientific_name），但其实也可以在可能产生冲突时才处理。

在上一条 SQL 语句中，我们还定了一个用于保存该目的典型照片的列。这样我们就可以将该目最具代表性的鸟种或含有该目的多个鸟种的照片放进去。注意该列的类型被定为 BLOB。BLOB 这个词很有趣，而且朗朗上口，其实它是*二进制大对象*的意思。我们可以将一张图像文件，如 JPEG 文件，放进 BLOB 列。但这种做法通常是不好的，因为它会使得表变大，导致备份困难。更好的做法是将图像文件放在文件系统中，然后将其文件路径或 URL 地址存进数据库，以指示如何找到该文件。不过我还是定了 BLOB 列，以便让你知道有这种做法。

在定义完列以后，我们还加上了默认字符集和校对方式的定义。因为有些名称可能包含有 latin1 以外的字符，所以这里我们使用了 UTF-8（UCS Transformation Format，8-bit）。比如说，若我们的观鸟网站需要显示德文，那么 brief_description 列就需要能接受德语变音字母（如 ä），而 utf8 就能做到。

对于一个真的观鸟网站来说，bird_families 和 bird_orders 都应该要有更多的列才对，而且需要的表也不只我们建的这几个。但作为演示，现在这样就够了。

4.5 小结

建表时你可以指定更多选项。例如存储引擎就有多种选择。本章略有涉及，但那只是冰山一角。有些存储引擎可以让你的数据分区存放在服务器硬盘上的不同位置。不同的存储引擎对性能有不同的影响。有些选项和设定很少被指定，但它们确实有存在的理由。对目前来说，这方面我们已经讲得够多了。

本章中的一些内容甚至超出了这个阶段的学习范畴，尤其是参考表，即 bird_families 和 bird_orders。后面你将会深刻体会到它们的作用。第 5 章会理清一些关于表的问题，还会教你怎样更改表。当中还穿插着一些插入和选择数据的示例。在继续学习之前，请先做完下一节的习题。这将有助于你更好地理解表的原理和用途。

4.6 习题

除了本章中你键入 MySQL 服务器的那些语句，这里还提供了一些让你巩固建库建表技巧的练习题。有些题目要求你建的表，会在后面的章节里用到，所以，请务必完成下面的习题。

(1) 使用 DROP TABLE 语句将之前建的 bird_orders 表删掉。找回其建表语句。复制或键入文本编辑器，然后进行修改：将 brief_description 改成 TEXT 类型。注意不要留有多余的逗号。完成后，将改过的语句复制到 mysql 监视器上，并按 Enter 键来执行它。

如果报错，那就看看报错信息（可能不太明确），再看看编辑器上的语句。检查一下改动了什么，有没有改错。确认关键字和值的位置正确，而且没有错别字。改完后再试着执行，直至成功为止。

(2) 之前提过，我们还可以为每种鸟存储更多相关方面的信息。但不要将这些数据放进 birds 表，相反，再创建一个表，即参考表。用 CREATE TABLE 来创建，命名为 birds_wing_shapes。建三个列。第一列名为 wing_id，类型为 CHAR，最大长度是 2。设该列为 UNIQUE 键，但不带有 AUTO_INCREMENT。我们将手工输入两位代码来识别每行的数据——因为表中可能只有六行数据，所以手工输入是可以接受的。第二列叫 wing_shape，类型为 CHAR，最大长度是 25，用于描述鸟翼类型（例如锥形）。第三列是 wing_example，BLOB 类型，用于存储该种鸟翼形状的图片。

(3) 建完 birds_wing_shapes 表后，对它执行 SHOW CREATE TABLE 两次：一次以分号结尾，一次以 \G 结尾。看看哪种样式的结果更具表现力。

将返回的 CREATE TABLE 语句复制粘贴进文本编辑器。然后用 DROP TABLE 删掉 birds_wing_shapes。

在编辑器中改改复制来的语句。首先是存储引擎，如果 ENGINE 的值不是 MyISAM，则改成 MyISAM。接着，将字符集和校对方式分别改成 utf8 和 utf8_general_ci。

然后将改过的语句粘贴到 mysql 监视器上，并按 Enter 键来执行它。如果报错，那就看看那迷惑人的报错信息，再看看你编辑器上的语句。检查一下改动了什么，有没有改错。确认关键字和值的位置正确，而且没有错别字。改完后再试着执行，直至成功。一旦成功，就用 DESCRIBE 来查看该表长什么样。

(4) 再建两个类似 birds_wing_shapes 的表。一个用于存储关于鸟的身形的信息，另一个用于存储关于鸟嘴形状的信息。它们会有助于观鸟爱好者找鸟。给它们分别起名为 birds_body_shapes、birds_bill_shapes。

birds_body_shapes 表的第一列是 body_id，其类型为 CHAR(3)，并且是 UNIQUE 键。第二列是 body_shape，类型为 CHAR(25)。第三列是 body_example，设为 BLOB 类型，以便存放鸟的形状图像。

对 birds_bill_shapes 表，也建三个类似的列：类型为 CHAR(2)、作为 UNIQUE 键的 bill_id；CHAR(25) 类型的 bill_shape；存放鸟形图像的、BLOB 类型的 bill_shape。这两个表的 ENGINE 都填 MyIASM，而 DEFAULT CHARSET 填 utf8，COLLATE 则填 utf8_general_ci。都建完后，用 SHOW CREATE TABLE 检查一下你的工作。

第 5 章

更改表

即使计划得再好，你偶尔还是会需要更改表的结构或者其他某些方面的东西。你无法想到以后会对一个表做的所有事情，或向其输入什么样的数据。不过，更改表也不是什么难事。所以，你无需在建表时追求完美，而应该把表看成一个不断变化的事物。也许*表结构*这个词限制了你的思维：*表*和*结构*给人一种规范的感觉。为了让你消除这种感觉，我想用一句改编的老话来“刷新”你对表结构的认识：它不是用石头或者木头造的，而是用数码造的，所以它能随意变换。这话可能不会成为名言，但确实是有道理的。

本章会探索各种更改表的方法：如何增加或删除列，如何改变列的数据类型，如何增加索引，以及如何改变表或列的选项。另外，还会介绍一些改表的注意事项和潜在的数据问题。

5.1　改表需谨慎

在改表之前，尤其是改含有数据的表之前，应该做好数据备份。无论你的改动有多小，都应该这么做。如果你改了列的大小，有可能会丢失部分数据。如果将列的数据类型改成与之前的不兼容（例如，将字符串数据类型改成数值数据类型），有可能会丢失全部数据。

如果只改一个表，你可以在同一个数据库中将该表复制一份作为备份，以便在出错后能把表恢复回原本的样子。更好的做法是，在复制出的表上做改动。甚至，你还可以将副本放入 `test` 数据库，然后在 `test` 中改表。最后，用改好的表替换原本的表。关于这种做法，本章后面会详细讲解。

不过，最好的做法还是使用 `musqldump` 工具来备份你要改的表，或备份整个数据库。这个工具会在第 14 章介绍。但为了让你更好地理解我在说什么，这里举个例子：在命令行——不是 `mysql` 客户端——输入以下命令，即可使用 `mysqldump` 备份 `birds` 表。（你需要

有备份文件所存放目录的读和写权限，本例中该目录是 /tmp，但你最好用别的目录，即只有你能进入，而 `mysql` 用户又能读写的目录。）

```
mysqldump --user='russell' -p \
rookery birds > /tmp/birds.sql
```

如你所见，首先在第一行指定用户名（请填你的用户名），它需要用单引号或双引号括起来，然后用 `-p` 来让 `mysqldump` 提示你输入密码。`mysqldump` 还有很多其他选项，但对于我们的需求，现在这些就够了。顺便说一下，这个命令可以写成一行，也可以写成多行（就像本例，用反斜杠告知 shell 下一行还有）。而在第二行，我们指定了数据库名和表名。接着是重定向符号（`>`），它告诉 shell 将 `mysqldump` 的结果送到 /tmp 目录的 birds.sql 文件中。

上例只备份了 `birds` 表。我们最好还是备份整个 `rookery` 数据库。用 `mysqldump` 的话，可在命令行输入：

```
mysqldump --user='russell' -p \
rookery > rookery.sql
```

备份整个 `rookery` 是有意义的，因为这有助于你做练习时误删表后再恢复。另外，在每章结束后，都备份一次 `rookery`，也是不错的做法。每次备份，都将备份文件按章节命名（如 rookery-ch1-end.sql、rookery-ch2-end.sql 等），这样你就能轻松地回顾本书的各个阶段。

后面如果出了问题，你想回到某章，可以在命令行输入：

```
mysql --user='russell' -p \
rookery < rookery-ch2-end.sql
```

注意这里不是用 `mysqldump`。恢复备份文件需要用 `mysql`。当备份文件（rookery-ch2-end.sql）被读进数据库时，会先删除 `rookery` 的表和数据，再以备份的来重置。于是，没有备份的数据都将会丢失。注意，恢复命令用的是另一个重定向符号——小于号（`<`），来告诉 `mysql` 从文本文件 rookery-ch2-end.sql 中读取内容。我们可以指定只恢复备份文件中的某个表，或设定其他限制，只恢复某些东西。具体操作会在第 14 章介绍。下面就来学习在 MySQL 和 MariaDB 中改表的必修技能。

5.2　必修的改表技能

在使用表的过程中，无论是刚建好时，还是输入数据后，你都很可能会想要更改它：增加一列，改变列的数据类型（例如让它容纳更多字符），修改列名使其意义更明确，或者令某列与其他表的列能更好地关联。你可能会想增加或修改索引，以加快数据查找速度。你还可能想改变默认值或选项。而所有这些操作，都可通过 `ALTER TABLE` 实现。

`ALTER TABLE` 的基本语法很简单：

```
ALTER TABLE table_name changes;
```

其中 `table_name` 请填成你要改的表名，`changes` 则填成具体的更改命令。各种可用的更改命令都会在本章依次教授。

这条 SQL 语句本身很简单，麻烦的是各种更改命令。事实上，造成这种麻烦的根本原因

是，ALTER TABLE 并不是天天都会用到。一般人改表，就是从书本或文档中查找更改命令，将其输入到服务器，然后忘记自己输了什么。相反，插入数据和查询数据的 SQL 语句（INSERT 和 SELECT）经常用，所以我们对其语法非常熟悉。所以，人们不记得如何用 ALTER TABLE 改表是很正常的。

最常做的改表操作之一就是增加列。其做法是，在上例的 changes 处，使用 ADD COLUMN 子句。为了演示，我们给 bird_families 表增加一列，使其能与 bird_orders 表关联。这两个表我们在第 4 章建过了。新加的列叫 order_id，与 bird_orders 中的一样。这样命名是可以的，而且说不定还有好处。具体来说，在 mysql 输入以下命令：

```
ALTER TABLE bird_families
ADD COLUMN order_id INT;
```

这太简单了。给该表加了一个名为 order_id 的、能容纳整数的列，不过我们没有让它像 bird_orders 的 order_id 那样自增。我们不需要它自增，因为我们只会在 bird_orders 中插入新的 order_id，而在 bird_families 中参考 bird_orders。

再举一个 ADD COLUMN 的例子：给 birds 表增加两列，使其能与第 4 章习题中所建的两个表（即 birds_wing_shapes 和 birds_body_shapes）关联。在操作之前，先把 birds 复制一份，然后在副本上操作。最后，用改好的表替换原表。

要复制 birds 表，我们会用 CREATE TABLE 语句（第 4 章讲过）和 LIKE 子句，并且将新表建在 test 数据库中以与原表区分开来（这不是必要的，但把一个测试用的数据库与正式的区分开来是好的做法）。你需要在 mysql 中输入如下命令：

```
CREATE TABLE test.birds_new LIKE birds;
```

接着输入以下两行，切换客户端的默认数据库，并查看新建的那个表：

```
USE test

DESCRIBE birds_new;
```

DESCRIBE 会展示新表的表结构。因为我们只复制了 birds 的表结构，所以新表是没有数据的。为使其有数据，可用 INSERT 加 SELECT：

```
INSERT INTO birds_new
SELECT * FROM rookery.birds;
```

这是没问题的。不过，我们还有另一种方法，可以在建表时同时复制表结构和数据：

```
CREATE TABLE birds_new_alternative
SELECT * FROM rookery.birds;
```

这样就建立了 birds_new_alternative 表，并在其中填好了数据。然而，当你使用 DESCRIBE 来查看该表时，就会发现其 bird_id 列没有设为 PRIMARY KEY 和 AUTO_INCREMENT。如果你要把所有设定也复制过来，那么第一种做法，即 CREATE...LIKE 然后再 INSERT...SELECT 会比较好。所以，删掉那个 birds_new_alternative 吧：

```
DROP TABLE birds_new_alternative;
```

用 DROP TABLE 语句时要小心。一旦删除一个表，通常没有方法（至少没有轻松的方法）把表找回，除非你有数据库的备份。这就是为什么我在本章开头建议你做好备份。

现在让我们更改新表，给它加个 wing_id 列，使其能与 birds_wing_shapes 表连接。在 mysql 中输入以下命令：

```
ALTER TABLE birds_new
ADD COLUMN wing_id CHAR(2);
```

这会增加一个名为 wing_id 的列到该表上，它是定长字符类型，最多容纳两个字符。我们得确保它的类型和大小都跟 birds_wing_shapes 的对应列一致，这样才能用这两列来连接两个表。

然后，看看 birds_new 的结构。在 mysql 中输入以下命令：

```
DESCRIBE birds_new;

+-----------------+--------------+------+-----+---------+----------------+
| Field           | Type         | Null | Key | Default | Extra          |
+-----------------+--------------+------+-----+---------+----------------+
| bird_id         | int(11)      | NO   | PRI | NULL    | auto_increment |
| scientific_name | varchar(100) | YES  | UNI | NULL    |                |
| common_name     | varchar(50)  | YES  |     | NULL    |                |
| family_id       | int(11)      | YES  |     | NULL    |                |
| description     | text         | YES  |     | NULL    |                |
| wing_id         | char(2)      | YES  |     | NULL    |                |
+-----------------+--------------+------+-----+---------+----------------+
```

结果中的前五列你应该都还记得吧。它们都来自我们在第 4 章中建的 birds 表。多出来的就是我们刚刚加的。现在注意 wing_id 列，它被加到了末尾。MySQL 和 MariaDB 是不关注列的排位的，但是，我们很关注，尤其是面对那些拥有大量列的表时。所以，我们重新增加 wing_id，并且这次要告诉 MySQL 把它放在 family_id 后面。首先，删掉刚加的这列。因为它是新列，所以不用担心丢失数据。

```
ALTER TABLE birds_new
DROP COLUMN wing_id;
```

这比增加列还要简单。注意我们没有提及列的数据类型或其他选项，因为删列不需要知道这些。DROP COLUMN 会删掉一列以及该列的所有数据。MySQL 和 MariaDB 是没有 UNDO 命令的，所以在生产环境中操作时要小心。

现在来重新增加一次：

```
ALTER TABLE birds_new
ADD COLUMN wing_id CHAR(2) AFTER family_id;
```

这使 wing_id 列加在了该表的 family_id 列后面。你可再用 DESCRIBE 来查看结果。顺便说一句，若想将新列加在最前面，要用 FIRST，而不是 AFTER。FIRST 后不需要带列名。

ALTER TABLE 的 ADD COLUMN 子句还可以用来一次增加多列，并指定这些列的排位。我们这就来试试增加三列。其中会有分别用于跟第 4 章习题所建的 birds_body_shapes 表和 birds_bill_shapes 表关联的列。另外我们还会多加一个域，来记录某一种鸟是否是濒危物

种。在增加列的同时，我们还想改变 common_name 列的宽度。现在它是 50 个字符的宽度，可能不足以容纳那些俗名很长的鸟种。而修改列的大小，就需要用到 CHANGE COLUMN 子句。在 mysql 中输入以下命令：

```
ALTER TABLE birds_new
ADD COLUMN body_id CHAR(2) AFTER wing_id,
ADD COLUMN bill_id CHAR(2) AFTER body_id,
ADD COLUMN endangered BIT DEFAULT b'1' AFTER bill_id,
CHANGE COLUMN common_name common_name VARCHAR(255);
```

这里 ADD COLUMN 的用法与之前的 ALTER TABLE 相似。其中有些差异是需要注意的。首先，这次我们输了三个 ADD COLUMN，两两之间以逗号分隔。你可能会觉得，一个 ADD COLUMN 带着多个逗号分隔的列声明应该是可以的。这种想法很常见，甚至有些老手也会这样以为，但其实这是错的。你可以在 ALTER TABLE 中加入多个子句，但每个子句只能指定一列。这个限制看起来没有必要，但因为改表影响重大，如果输错内容，可能后果严重，所以为了让你做好检查，就多写几遍子句吧。

在我们所加的 endangered 列中，使用了本书之前还未用过的 BIT 数据类型。这个类型只占一位，状态有两种：1 代表有设值，0 代表没设值。我们会用它来表示该物种是否濒危。注意这里我们用 DEFAULT 指定了该列的默认值。还有，为了给位类型设值，我们需要将值用引号包围，并在前面加上字母 b。这种数据类型有个问题。它确实有保存值，但就是显示不出值。如果有设值，则查询时不会显示任何东西，导致 ASCII 形式的结果集会在该位置有一个向左的缩进。这是 MySQL 的一个 bug，不过以后肯定会修复的——可能在你读这本书时已经修好了。这个 bug 对我们的使用没什么大影响。在导入数据之后，我们就会看看它实际的样子。

而在 CHANGE COLUMN 子句中，我们输入了 common_name 两次。第一次用于指示我们要改哪一列。第二次用于给该列指定一个新名字。不过即使没打算改列名，你还是要写上一个名字，否则会报错，并拒绝执行。接下来指定数据类型。即使你只打算改名而没打算改类型，你还是要写上一个类型。基本上，在使用 CHANGE COLUMN 时，就算你只想修改该列的某一方面，服务器也需要你完整地声明整个新列。

还有一点要注意的是，我们这里对 AFTER 的使用与之前的不太一样。对于新增的第二列，即 bill_id，我们要求把它加在 body_id 后面。你可能会认为，因为 body_id 的声明也是在这同一条语句之中，所以这会报错。然而，MySQL 是按 ALTER TABLE 的子句的编写顺序来执行整个操作的。在有些版本中，它会建一个临时的副本表，然后在该副本上按子句的声明顺序（即从左至右，而对于我们的例子，则是从上至下），陆续执行它们。都执行完后，如果没有错误，它就会用这个副本来替换原表，这有点像我们手工的做法，只不过它是在后台运作的，而且它更快速。

如果处理 ALTER TABLE 语句的任何一个子句时出错，它就会删掉那个副本表，而毫不影响原表，并返回错误信息给客户端。于是在上例中，MySQL 会创建一个临时表，接着它首先会增加 body_id 列，再在其后面增加 bill_id 列。你可能试过在所有 ADD COLUMN 后都使用 AFTER wing_id。当然这不会报错，但是这些列的位置就与声明时的顺序反过来了（即最后会变成 wing_id，endangered，bill_id，body_id）。所以，如果要把 body_id 放在 wind_id 后面，而 bill_id 放在 body_id 后面，那么我们就要如这个例子般编写语句。

下面来试试改变 endangered 列的值。该表目前只有五行，而且没有一行是被标记为濒危的。现在将其中四行的值设成 0。我们会用到 UPDATE 语句（如果你觉得它很陌生，也不用担心，这会在第 8 章详解）：

```
UPDATE birds_new SET endangered = 0
WHERE bird_id IN(1,2,4,5);
```

这样就将括号中列出的 bird_id 对应的行的 endangered 设置为 0 了，或者说，将 endangered 取消设置了。简单来说，就是只有 bird_id 为 3 的没改，其他的都改了。回忆一下建 endangered 时，我们曾为其指定默认值 b'1'，即默认设置。而上例则是把 WHERE 中提及的四行取消设置。

现在我们根据 endangered 来用 SELECT 语句获取数据（SELECT 在第 3 章和第 7 章有介绍）。因为现在 birds_new 列变多了，所以我们用 \G 模式输入以下 SQL 语句，以便阅读。

```
SELECT bird_id, scientific_name, common_name
FROM birds_new
WHERE endangered \G

*************************** 1. row ***************************
        bird_id: 3
scientific_name: Aix sponsa
    common_name: Wood Duck
*************************** 2. row ***************************
        bird_id: 6
scientific_name: Apteryx mantelli
    common_name: North Island Brown Kiwi
```

这个 WHERE 子句的意思是，只选取 endangered 有值的行。对于 BIT 类型的列来说，endangered 的意思等同于 endangered = 1。如果想选取没设置 endangered 的行，则可使用 NOT 运算符：

```
SELECT * FROM birds_new
WHERE NOT endangered \G
```

看过 Wood Duck 和 Kiwi 鸟后，可能你希望 endangered 列能表示更多值，因为濒危也是有分各种级别的。我们可以（也应该）另外建一个表来作为濒危级别的参考表，但为了让你能学到改表的技能，我还是通过改列来做。与此同时，我还会演示如何将列重新排位。为了达到这个目的，我们需要一个新的子句 MODIFY COLUMN：

```
ALTER TABLE birds_new
MODIFY COLUMN endangered
ENUM('Extinct',
     'Extinct in Wild',
     'Threatened - Critically Endangered',
     'Threatened - Endangered',
     'Threatened - Vulnerable',
     'Lower Risk - Conservation Dependent',
     'Lower Risk - Near Threatened',
     'Lower Risk - Least Concern')
AFTER family_id;
```

注意 MODIFY COLUMN 子句的语法只要求你输入一次列名，因为 MODIFY COLUMN 是不能用来改列名的。要改列名，得用 CHANGE COLUMN 子句。这里，我们还使用了一种新的数据类型——ENUM，它使我们能枚举出可接受的值。这些值需要放在一对括号中，每个值用引号包围，各值之间以逗号分隔。

接着，用 SHOW COLUMN 语句加上 LIKE 子句，以使结果仅显示 endangered 列的设定：

```
SHOW COLUMNS FROM birds_new LIKE 'endangered' \G

*************************** 1. row ***************************
  Field: endangered
   Type: enum('Extinct','Extinct in Wild',
              'Threatened - Critically Endangered',
              'Threatened - Endangered',
              'Threatened - Vulnerable',
              'Lower Risk - Conservation Dependent',
              'Lower Risk - Near Threatened',
              'Lower Risk - Least Concern')
   Null: YES
    Key:
Default: NULL
  Extra:
```

结果中除了列举出各个值外，还告诉我们，NULL 是允许的，并且也是默认值。我们可以通过 NOT NULL 子句来禁止 NULL 值。

如果你想再增加多个值，可以再次使用 ALTER TABLE 和 MODIFY COLUMN，而不带上 AFTER——除非你又想改变该列的位置。除了新增加的值，你还需要再次枚举出原本所有的值。

想要使用枚举值的话，你可以完整输入该值，又或者如果你知道这些枚举值的顺序，可以用序号来引用某个值。第一个枚举值的序号就是 1。举个例子，你可以用如下命令来把该表的所有鸟都设为 Lower Risk - Least Concern，即第七个枚举值：

```
UPDATE birds_new
SET endangered = 7;
```

之前说过，如果可能的值不多，那么 ENUM 类型可用于代替参考表。然而，我们这里定的 endangered 的值不算短但又不够详细。如果我们需要展示更详细的信息，可以再建一个参考表以作补充。这个参考表会有与这些枚举值一一对应的行，并且每行还有其他列，以显示详尽的相关信息。这样做的话，birds 表的枚举值就可以设计得更简单（例如，Lower Risk - Least Concern 改成 LR-LC），而该值的具体解释就放在另建的参考表中。

不过，在数据操作时使用枚举值的序号，直接把枚举列当成参考表，这样似乎更简单。尽管如此，我们还是应该再改改该列，并新建一个参考表。我们会在后面具体操作。

动态列

既然讲到 ENUM 类型，那就暂时把 ALTER TABLE 放一边，继续延伸一下，讲讲动态列，这是 MariaDB 的 v5.3 才有的东西。它与 ENUM 类似，但不是一堆枚举值，而是一堆键值对。第

一次听说可能会觉得很困惑，但看过一些例子后你就会明白了。所以，先来建一些有动态列的表吧。

为了使观鸟爱好者网站更吸引人，我们决定对爱好者们做一些调查。先从简单的开始，我们会让他们选出自己最喜欢的鸟。过段时间后，我们可能还想让他们选出最佳观鸟地点，或者最喜爱的望远镜制造商和模型。于是，我们需要建一些表来实现这个需求。

如果你用的不是 MariaDB，也没打算用 MariaDB 来替换 MySQL，那么你可以不用动手，仅仅读书就行了。如果你安装了 MariaDB，那请输入以下代码：

```
USE birdwatchers;

CREATE TABLE surveys
(survey_id INT AUTO_INCREMENT KEY,
survey_name VARCHAR(255));

CREATE TABLE survey_questions
(question_id INT AUTO_INCREMENT KEY,
survey_id INT,
question VARCHAR(255),
choices BLOB);

CREATE TABLE survey_answers
(answer_id INT AUTO_INCREMENT KEY,
human_id INT,
question_id INT,
date_answered DATETIME,
answer VARCHAR(255));
```

第一个表用于存放调查问卷清单。第二个表则用于存放问卷里的问题。因为我们只需受访者投票，所以 choices 列将会包含各种选项。这列使用一个非常通用的类型——BLOB，但实际上它将用于动态列。所选的数据类型必须能容纳动态列实际的数据。而 BLOB 就是一个不错的选择。

第三个表用于存放问题的回答。这次我们的动态列使用 VARCHAR 类型。最后 survey_answers 会以 question_id 跟 survey_questions 关联，survey_questions 会以 survey_id 跟 surveys 关联。

接着，在这些表中输入一些数据。如果你用的是 MariaDB，则可用以下 SQL 语句：

```
INSERT INTO surveys (survey_name)
VALUES("Favorite Birding Location");

INSERT INTO survey_questions
(survey_id, question, choices)
VALUES(LAST_INSERT_ID(),
"What's your favorite setting for bird-watching?",
COLUMN_CREATE('1', 'forest', '2', 'shore', '3', 'backyard') );

INSERT INTO surveys (survey_name)
VALUES("Preferred Birds");

INSERT INTO survey_questions
```

```
(survey_id, question, choices)
VALUES(LAST_INSERT_ID(),
"Which type of birds do you like best?",
COLUMN_CREATE('1', 'perching', '2', 'shore', '3', 'fowl', '4', 'rapture') );
```

于是我们有了两个问卷：一个问人们喜欢到哪里观鸟，另一个则问人们喜欢哪种鸟。这里我们只提供一些很简单的、不全面的选项。COLUMN_CREATE() 用于创建选项的枚举：每个选项都有一个键和一个值。例如，选项 1 是 forest，选项 2 是 shore，选项 3 是 backyard。而从 MariaDB v10.0.1 开始，键还可以用字符，而不只是数字。

下面来看看如何从动态列获取数据：

```
SELECT COLUMN_GET(choices, 3 AS CHAR)
AS 'Location'
FROM survey_questions
WHERE survey_id = 1;

+----------+
| Location |
+----------+
| backyard |
+----------+
```

这个命令会返回第三个选项。我们使用了 COLUMN_GET()，其中第一个参数填动态列的列名，第二个参数填该动态列的键，这样就能获取其对应的值了。同时，我们还包含了一个 AS 来将返回的数据转换成我们指定的类型（即 CHAR）。

接着，录入会员的答案。如果你在看的是这本书的电子版，那么把下面的语句复制粘贴到 MariaD 服务器就好了：

```
INSERT INTO survey_answers
(human_id, question_id, date_answered, answer)
VALUES
(29, 1, NOW(), 2),
(29, 2, NOW(), 2),
(35, 1, NOW(), 1),
(35, 2, NOW(), 1),
(26, 1, NOW(), 2),
(26, 2, NOW(), 1),
(27, 1, NOW(), 2),
(27, 2, NOW(), 4),
(16, 1, NOW(), 3),
(3, 1, NOW(), 1),
(3, 2, NOW(), 1);
```

这里没有多少行，但暂时来说已经足够了。现在就来给第一个问卷点票：

```
SELECT IFNULL(COLUMN_GET(choices, answer AS CHAR), 'total')
AS 'Birding Site', COUNT(*) AS 'Votes'
FROM survey_answers
JOIN survey_questions USING(question_id)
WHERE survey_id = 1
AND question_id = 1
GROUP BY answer WITH ROLLUP;
```

```
+--------------+-------+
| Birding Site | Votes |
+--------------+-------+
| forest       |     2 |
| shore        |     3 |
| backyard     |     1 |
| total        |     6 |
+--------------+-------+
```

在 `WHERE` 子句中，我们指定了 `survey_id` 和 `question_id` 来选取我们想看的问卷和问题。然后我们取出该问题的所有答案，把它们聚集起来，计算出每个答案被投了多少票。

不过这里没多少数据。我会添加更多答案，以造出一个更大的表来再做统计。你可以到我的网站（http://mysqlresources.com/files）下载这个表。本书后面的示例会用到它。动态列是一个新特性，它仍在不断开发中，所以暂时来说，简单介绍已足够。下面我们回到更正式的改表话题。

5.3　选修的改表技能

`ALTER TABLE` 最常用于增加和重命名列，但除此之外，它还可以用于设置表和列的某些选项。你还可以用它来设置表变量，以及列的默认值。本节会介绍如何修改这些设定和值，以及如何重命名一个表。另外，索引也是可以更改的。这个话题会在 5.4 节展开。

5.3.1　设置列的默认值

你可能注意到了，之前那些 `DESCRIBE` 语句的结果示例里都有 `Default` 列。你可能还注意到了，几乎所有域的默认值都是 NULL。这意味着，如果用户不给该列输入一个值，那么该列的值将是 NULL。如果你想为某列指定默认值，那么可以在建表时声明。而对于已建好的表，可以使用 `ALTER TABLE` 语句来指定一个有别于 NULL 的默认值。这并不会改变现有行的值——包括那些以前套用了默认值的行。改默认值可用 `CHANGE` 子句或 `ALTER` 子句。我们先来看看一个使用 `CHANGE` 子句的示例。

假设在我们的数据库中，大部分鸟的 `endangered` 列的值都是 `Lower Risk - Least Concern`。那么我们可以直接更改该列的默认值，这样就不用每次都在 `INSERT` 语句（用于插入数据）中输入 `Lower Risk - Least Concern` 或其对应的枚举序号了。所以我们接下来就来操作，并且，为了适配将建立的用于存放鸟种保护状态的参考表，我们还要把 `endangered` 列从 `ENUM` 类型改成 `INT`。另外，为了更加好玩，我们会在建表后，将之前输过的 `endangered` 值插入进去。下面我们就开始了，先在 `mysql` 中输入以下命令，创建参考表。

```
CREATE TABLE rookery.conservation_status
(status_id INT AUTO_INCREMENT PRIMARY KEY,
conservation_category CHAR(10),
conservation_state CHAR(25) );
```

参考表被命名为 `conservation_status`，这比 `endangered` 听起来更明了。注意我们将每种状态都分成两列。例如 `Lower Risk - Least Concern`，它其实指的是 `Lower Risk` 类中的

Least Concern 状态。所以我们需要有两列，Lower Risk 放在 conservation_category，而 Least Concern 放在 conservation_state。

接着，将所有的数据都插入该表中。这会用到 INSERT 语句（第 3 章简单介绍过）：

```
INSERT INTO rookery.conservation_status
(conservation_category, conservation_state)
VALUES('Extinct','Extinct'),
('Extinct','Extinct in Wild'),
('Threatened','Critically Endangered'),
('Threatened','Endangered'),
('Threatened','Vulnerable'),
('Lower Risk','Conservation Dependent'),
('Lower Risk','Near Threatened'),
('Lower Risk','Least Concern');
```

如果你看不太懂上面的语句，先别管它，输进去就好了，我们会在第 6 章深入讲解。然后，我们来看看这个表有了数据以后是什么样子。输入以下 SQL 语句（粗体部分），不要把结果也输入进去。

```
SELECT * FROM rookery.conservation_status;

+-----------+-----------------------+------------------------+
| status_id | conservation_category | conservation_state     |
+-----------+-----------------------+------------------------+
|         1 | Extinct               | Extinct                |
|         2 | Extinct               | Extinct in Wild        |
|         3 | Threatened            | Critically Endangered  |
|         4 | Threatened            | Endangered             |
|         5 | Threatened            | Vulnerable             |
|         6 | Lower Risk            | Conservation Dependent |
|         7 | Lower Risk            | Near Threatened        |
|         8 | Lower Risk            | Least Concern          |
+-----------+-----------------------+------------------------+
```

第一列取自 AUTO_INCREMENT，这是我们在建表时指定的，而其他列，是我们 INSERT 时指定的。

注意我们在表名前还加了数据库名（即 rookery.conservation_status）。这是因为我们用 USE 把默认数据库设为了 test。接着回到 birds_new 表，我们已经准备好改 endangered 列了。之前已决定好，我们需要将该列的默认值设为 Lower Risk - Least Concern，或 conservation_status 表中相应的列组合所确定的 status_id。看看 conservation_status 的查询结果，你会发现我们所需的 status_id 是 8。现在，就用以下语句来修改列名和默认值：

```
ALTER TABLE birds_new
CHANGE COLUMN endangered conservation_status_id INT DEFAULT 8;
```

这个语法基本上与本章之前使用 CHANGE 子句的例子一致（即是那个输入两次列名然后再改数据类型的例子，不过那次你没想要改列名）。不同的是，这次我们还增加了关键字 DEFAULT，并在其后带上我们要设置的默认值——如果该值是字符串，则需要用引号包围。本例也改了列名。但如果你只想设置默认值，可以使用 ALTER 子句。我们就试试用它来设

置默认值为 7：

```
ALTER TABLE birds_new
ALTER conservation_status_id SET DEFAULT 7;
```

这更简单了。它只设置了默认值。注意第二行开头用的是 ALTER 而非 CHANGE。其后接列名，然后是属于 ALTER 的 SET 子句。现在用 SHOW COLUMNS，单单显示该列：

```
SHOW COLUMNS FROM birds_new LIKE 'conservation_status_id' \G

*************************** 1. row ***************************
  Field: conservation_status_id
   Type: int(11)
   Null: YES
    Key:
Default: 7
  Extra:
```

如你所见，现在默认值是 7。如果我们又改变主意，想将其设回 NULL 或该列数据类型的默认值，那么可以用以下命令：

```
ALTER TABLE birds_new
ALTER conservation_status_id DROP DEFAULT;
```

关键字 DROP 的这种特殊用法不会删除列的数据，只会更改列的默认值设定。请在你的机器上用 SHOW COLUMNS 看看默认值重置的样子。然后，将它改回 7。

5.3.2 设置AUTO_INCREMENT的值

数据库中很多主表的主键都会用到 AUTO_INCREMENT 选项。它会在 information_schema 库的 tables 表中创建一个 AUTO_INCREMENT 变量。你应该记得这个数据库名。我们在 3.3 节中使用 SHOW DATABASE 语句时看到过它。每次你建表时，MySQL 都会在 information_schema 库的 tables 表中加入一行。而 tables 表中就有一列叫作 auto_increment。新增行时所需的自增值就从那里获取。除非你在建表时另外指定初始值，不然它初始就是 1。现在用 SELECT 来看看该列的值：

```
SELECT auto_increment
FROM information_schema.tables
WHERE table_name = 'birds';

+----------------+
| auto_increment |
+----------------+
|              7 |
+----------------+
```

因为我们没有在建 birds 表时设置 AUTO_INCREMENT 的值，而该表又只输入过六种鸟的数据，所以它的初始值是 1，现在增长到了 7。这意味着你为 birds 表插入下一行时，可从这里获取 7 作为 bird_id。

如果你想改变某个表的 AUTO_INCREMENT 的值，可以使用 ALTER TABLE。为了演示，现在试试将 birds 表的 AUTO_INCREMENT 的值设为 10。先切换回 rookery 库。在 mysql 中输入以下

命令：

```
USE rookery

ALTER TABLE birds
AUTO_INCREMENT = 10;
```

这样，birds 表的下一行的 bird_id 就能从 10 开始了。更改自增值并不常见，但你需要知道 ALTER TABLE 还有这种用法，毕竟增长一下见识是不错的。

5.3.3　改表和建表的另一种方法

有些时候你可能会觉得某些表建了太多列。或许有些列拿出来放在一个独立的表中会更好。又或者你在一个现有的表中加了一些列，但过段时间发现它的结构不太对。无论哪种情况，你都可以建一个小表，然后将大表的数据移过去。一种做法是，建一个表，其列的设置如同原大表中所需移动的列，然后从大表复制数据到小表，最后删掉大表中不再需要的列。如果你用这种做法来转移数据，那么必须保证新表的列的设定与原表一致，以避免数据丢失或其他问题。

更简单的做法是使用 CREATE TABLE 语句加 LIKE 子句，来根据原表创建出新表。现在试试建一个 birds 表的副本。在 mysql 中输入以下命令：

```
CREATE TABLE birds_new LIKE birds;
```

这就建了另外一个如同 birds 的表，它名为 birds_new。用 SHOW TABLES 就能看到这两个表。

> 表名含有下划线（即 _）是可以的，而连字符则最好不要。MySQL 会将连字符解析为减号，并用其两端的文字做运算，这会产生错误。如果你真的要在表名中用连字符，则无论何时都要用引号包围该表名。

用以下语句看看你有什么表和数据：

```
DESCRIBE birds;

DESCRIBE birds_new;

SELECT * FROM birds_new;
Empty set (0.00 sec)
```

前两句用于显示这两个表的结构。你会发现它们除了表名，其他完全一样。为了节省空间，我就不把结果贴上来了。

第三句用于显示 birds_new 表的所有数据。但因为我们在建新表时只从 birds 表复制了表结构，所以是没有数据的——它也返回了“空”的提示。如果想要有数据的话，可以在改完表后再复制数据。

此方法也可用于想要对表进行大改的情况。如果是大改的话，在表的副本上操作是不错的做法。用 ALTER TABLE 修改复制出来的表（如 birds_new），改完后，从旧表复制数据到新

表，最后删掉旧表，给新表改名。

不过，还有一个小问题。之前说过，它们除了表名，其他方面都一样，但这并不完全正确。还可能有一点不同的是，如果新表有一列使用 AUTO_INCREMENT 作为默认值，那么它自增值将是从 0 开始。你需要根据 birds 表 AUTO_INCREMENT 的当前值来设定新表 AUTO_INCREMENT 的初始值，以使新表的新行能获取正确的标识号。在 mysql 输入以下语句：

```
SHOW CREATE TABLE birds \G
```

结果（这里没贴出来）中的最后一行会显示 AUTO_INCREMENT 的当前值。例如这样：

```
...
) ENGINE=MyISAM AUTO_INCREMENT=6 DEFAULT CHARSET=latin1 COLLATE=latin1_bin
```

在此摘录中，可以看到 AUTO_INCREMENT 的当前值是 6。现在，用以下语句来将 birds_new 的值也改成跟它一样：

```
ALTER TABLE birds_new
AUTO_INCREMENT = 6;
```

当你准备好复制数据时，就可以用 INSERT...SELECT 语法了。这会在 6.3 节中介绍。

除了在改好新表后再复制数据，你还可以在创建新表的同时进行复制。如果你想将某些列连带数据一起复制，并且不打算修改这些列，那么这种做法很适合你。要达到这个目的，我们依然使用 CREATE TABLE，但语法稍有不同。

现在假设我们决定创建一个新表，用于保存每种鸟的详细信息（比如迁徙模式、习性，等等）。我们打算让新表拥有 birds 表的 description 列及其数据。也就是说，我们要创建一个新表，并把该列的设定和数据，以及 bird_id 复制进去。具体可在 mysql 中使用以下命令：

```
CREATE TABLE birds_details
SELECT bird_id, description
FROM birds;
```

这就基于 birds 表的两列，创建了含有相同列的 birds_details 表。与此同时，它还从 birds 表复制了那两列的数据到 birds_details 表。这里有一个细小的、但必须注意的区别，就是 AUTO_INCREMENT 的设定方式不同于之前的例子。现在用 DESCRIBE 来看看这一区别：

```
DESCRIBE birds_details;

+-------------+---------+------+-----+---------+-------+
| Field       | Type    | Null | Key | Default | Extra |
+-------------+---------+------+-----+---------+-------+
| bird_id     | int(11) | NO   |     | 0       |       |
| description | text    | YES  |     | NULL    |       |
+-------------+---------+------+-----+---------+-------+
```

区别在于，这里的 bird_id 没有使用 AUTO_INCREMENT。这是对的，我们就是要为每一行手工设置 bird_id 的值。不是每一种鸟都有详细信息，另外，亦无必要在录入 birds 表的同时录入 birds_details 表。我们确实可以为 birds_details 表的 bird_id 加上 AUTO_

INCREMENT，但这会造成问题——需要与 birds 表的录入顺序一致，这没有道理。如果只是为了让 birds_details 表的 bird_id 不重复，可以使用 ALTER TABLE 给其加上 UNIQUE。这就能使一种鸟只有一个描述，那挺好的。具体会在 5.4 节讲解。

CREATE TABLE...SELECT 语句建立了拥有两列的 birds_details 表。尽管如此，但我们还想要更多列来保存鸟的信息。所以，在本章结尾的习题中，我们会使用 ALTER TABLE 再增加一些列。现在，先用以下命令删掉 birds 表的 description 列：

```
ALTER TABLE birds
DROP COLUMN description;
```

这将删除该列及其中数据。所以，使用时请小心。该子句会在第 6 章详解。

5.3.4 重命名一个表

之前的几节介绍过怎样修改表里的列，其中包括重命名列。但有时候，可能你还想重命名表。原因可能有很多种，例如你想把列名改成更直白的名称。你甚至可以用它来进行表的置换，即删掉一个表，然后让另一个表使用被删表的名字。上一节中的一些例子就需要这种做法。

我们在 test 数据库中建立了 birds_new 表。我们计划修改该表，然后从 rookery 数据库删掉 birds 表，最后以 test 的 birds_new 表作为替代。为了完全地取代 birds 表，这里我们会将 birds_new 表重命名为 birds。这是 ALTER TABLE 无法做到的，因为它只能用于修改表结构，而不能重命名表。相反，RENAME TABLE 则做得到。不过先别开工。先看看以下这个重命名表的通用例子。看看就好，不要执行：

```
RENAME TABLE table1_altered
TO table1;
```

这条 SQL 语句会将 table1_altered 重命名为 table1。它假设数据库中没有一个叫 table1 的表。不过就算有，它也不会重写 table1，而只会给你一个错误信息，并让 table1_altered 维持原样。

RENAME TABLE 语句还可用于将表移到另一个数据库中。如果你在一个数据库中建了一个表，就像我们之前在 test 数据库中建了 birds_new 表那样，而现在又想将它放到另一个数据库中，那么这个命令就能帮上忙了。因为 RENAME TABLE 可以同时改表名和移动表，所以我们不是按照上例的语法来操作 birds_new。（顺便说一下，移动表而不改表名是可以的。你只需将新库名后接相同的表名。）在我们的例子中，你需要先删除或重命名 rookery 数据库中那个没被修改的表。通常给它重命名是比较安全的做法，所以我们就这么做。

先将 rookery 数据库中的 birds 表重命名为 birds_old，然后将 test 数据库中的 birds_new 表移到 rookery，并同时将其改名为 birds。要用一条 SQL 语句来完成这些改动，可以这样写：

```
RENAME TABLE rookery.birds TO rookery.birds_old,
test.birds_new TO rookery.birds;
```

如果在修改过程中出现问题，你会得到一个错误信息，并且不会实现任何修改。而如果一

切顺利，你就会在 rookery 数据库中拥有两个存放鸟种信息的表。

现在用 SHOW TABLES 来看看 rookery 数据库中有什么表。我们只想查询名字以 birds 开头的表，所以使用了 LIKE 子句和通配符 %。在 mysql 中输入：

```
SHOW TABLES IN rookery LIKE 'birds%';

+----------------------------+
| Tables_in_rookery (birds%) |
+----------------------------+
| birds                      |
| birds_bill_shapes          |
| birds_body_shapes          |
| birds_details              |
| birds_new                  |
| birds_old                  |
| birds_wing_shapes          |
+----------------------------+
```

其中 birds 表就是我们早先在 test 数据库中修改过的 birds_new 表，而原本的 birds 表已被重命名为 birds_old。结果中的其他表是本章前面建立的。因为它们的名字也是以 birds 开头，所以也出现在结果中。在用 SELECT 查过你没有丢失任何数据后，你可能就想删掉 birds_old 表了。删掉 birds_old 表需要用 DROP TABLE 语句。具体命令如下，但别输入它：

```
DROP TABLE birds_old;
```

5.3.5 重排序一个表

用于从表中获取数据的 SELECT 语句，可带有 ORDER BY 子句，用来使结果集有序显示。这在展示数据，尤其是大量数据时，很有帮助。有时我们也会想对表里的数据重新排序，尽管这不是很必要。你可以对那些几乎不做改动的表，如参考表，进行重排序。有时这会使得查询更快速，不过通常索引也能做到，而且还做得更好。

举个例子，如果你访问我的网站，会看到一个列举国家代码的表。我们可能会用这个表来跟网站的会员连接，又或者是在列出鸟种发现地时使用它。country_codes 表有一列用于存放两个字符的国家代码，还有一列用于存放国家名称。为了避免给每行会员或鸟种输入完整的国家名称，我们只输入国家代码（例如，打 us 以代表 United States of America）。该表已经按国家名称来排序了，但你或许会想让它按行的字母序来排。又或者你想在增加了一个争议地区的代码和名称后，使该表依然有序。

先看看这个表的数据是怎样的。在 mysql 中输入以下只显示前三行数据的 SELECT 语句：

```
SELECT * FROM country_codes
LIMIT 3;

+--------------+----------------+
| country_code | country_name   |
+--------------+----------------+
| af           | Afghanistan    |
| al           | Albania        |
```

```
| dz           | Algeria        |
+--------------+----------------+
```

如你所见，数据已经按 country_name 的值的字母序排列好了。现在用 ALTER TABLE 及其 ORDER BY 子句，使数据按 country_code 来重新排列。也许你并不是真的想这么排，但我们还是来做做，以便体验一下 ORDER BY 子句的效果。在改完之后，我们可以把它再改回来。在 mysql 中输入以下命令：

```
ALTER TABLE country_codes
ORDER BY country_code;
```

这应该很快能执行完。现在再次用 SELECT，并只显示前三行数据：

```
SELECT * FROM
country_codes LIMIT 3;

+--------------+----------------------+
| country_code | country_name         |
+--------------+----------------------+
| ad           | Andorra              |
| ae           | United Arab Emirates |
| af           | Afghanistan          |
+--------------+----------------------+
```

我们注意到，虽然没在 SELECT 中指定顺序，但结果不同了，现在这些行以 country_code 来排序。若要将它们改回按 country_name 来排序，可再次使用 ALTER TABLE，但指定 country_name 列，而非 country_code 列。

再说一次，重排序一个表几乎是没有必要的。因为我们还可以用 SELECT 再加上 ORDER BY 来排序：

```
SELECT * FROM country_codes
ORDER BY country_name
LIMIT 3;
```

这条 SQL 语句出来的结果与之前的 SELECT 一样，而且在速度上几乎没有差别。

5.4 索引

对于新手来说，ALTER TABLE 烦人的用法之一，就是用来修改索引了。如果你只用 ALTER TABLE 来重命名一个索引列，就会得到一个令人困惑的错误信息。例如，假设你想将 conservation_status 表的主键列 status_id 改名为 conservation_status_id。你可能会写这样的 SQL 语句：

```
ALTER TABLE conservation_status
CHANGE status_id conservation_status_id INT AUTO_INCREMENT PRIMARY KEY;

ERROR 1068: Multiple primary key defined
```

刚开始这么做时，你可能会以为自己记错了语法。于是你开始尝试其他组合，但最后无论怎样都不行。想避免这种错误，一次就做好，你需要对索引有更深入的理解，明白索引与

其所基于的列是分离的。

索引的作用就是令 MySQL 能快速地定位数据。它们的作用就像一本书后面的索引。对比以下各种在书中搜索内容的方法。例如，假设你想查找 `ALTER TABLE` 语法的相关内容，你可以从头开始，快速地翻页，一页页地查看——如果你看的是纸质版——直到找到 `ALTER TABLE` 的字眼。这就是不使用索引来查找数据。除此之外，你还可以翻看本书开头的目录，它就是一个很宽泛的索引，查找名为“更改表”的章节，然后在这些章节里，更进一步地在标题含有相关字眼的地方查找。这就是一个简单索引的例子。如果想更准确地查找，那么可以去到本书后面的索引，看看含有 `ALTER TABLE` 的书页有哪些，然后直接到这些页码中阅读。

MySQL 的索引与上面最后一个例子类似。如果没有索引，MySQL 就需要一行行去搜索。因为索引的体积更小，而且已经组织好以便快速遍历，所以 MySQL 可以通过它快速定位数据，然后直接跳到相应的行的位置。所以，创建表时，尤其是创建会拥有大量行的表时，请顺便创建索引。这样数据库会跑得更快。

记住了书本索引这个比喻，你应该能理解索引不是列，尽管它与列有关。为了在表中演示这一点，我们用 `SHOW INDEX` 语句来看看第 4 章所建立的 `humans` 表。在 `mysql` 中输入以下命令：

```
SHOW INDEX FROM birdwatchers.humans \G

*************************** 1. row ***************************
       Table: humans
  Non_unique: 0
    Key_name: PRIMARY
Seq_in_index: 1
 Column_name: human_id
   Collation: A
 Cardinality: 0
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
```

结果显示，有一个索引与 `human_id` 列相关（注意 `Column_name` 处）。`human_id` 不是索引，它是索引的根源。它的名字与索引的名字相同，而且索引也与这一列绑定，但它们绝不等同。现在我们改改这个表，并增加一个索引，来进一步理解这个概念。

假设用户有时会按会员的姓氏来检索 `humans` 表。如果没有索引，那么 MySQL 就会一行行地查找匹配的姓氏。我们可通过在 `SELECT` 前加上 `EXPLAIN`，来确认是否是这样。它会告知我们 `SELECT` 是基于什么来进行查找的，即向我们解释，在执行 `SELECT` 时，服务器做了什么——所以它不返回任何表内的数据，而是返回索引被如何利用的相关信息。在 `mysql` 中输入以下命令：

```
EXPLAIN SELECT * FROM birdwatchers.humans
WHERE name_last = 'Hollar' \G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: humans
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
        Extra: Using where
```

这里的 SELECT 语句在查 name_last 为 Hollar 的行的所有列，而 EXPLAIN 语句则对其进行分析。分析结果中，我们感兴趣的有 possible_key 列和 key 列——key 的意思是表按哪列进行索引。而 key 和索引一般是同一个意思。possible_key 表示 SELECT 语句本应用到的键。在我们这个例子中，name_last 是没有索引的。key 会列出语句实际用到的索引。而在本例中，它显示 NULL。因为现在表中只有四行，所以有没有索引都不会带来明显的性能区别。然而，如果有一天表中有了数千行数据，那么索引将大大提升查找人名的效率。

有时，用户不只会按姓来查询 humans 表，他们还会按名来查，甚至姓名一起查。为了对这些可能性有所准备，以及提升数据量增长后的查询性能，我们在这两列上建立索引。具体做法是，使用 ALTER TABLE 加 ADD INDEX 子句，就像这样：

```
ALTER TABLE birdwatchers.humans
ADD INDEX human_names (name_last, name_first);
```

接着，从 SHOW TABLE 语句的结果中看看该索引长什么样：

```
SHOW CREATE TABLE birdwatchers.humans \G

*************************** 1. row ***************************
       Table: humans
Create Table: CREATE TABLE `humans` (
  `human_id` int(11) NOT NULL AUTO_INCREMENT,
  `formal_title` varchar(25) COLLATE latin1_bin DEFAULT NULL,
  `name_first` varchar(25) COLLATE latin1_bin DEFAULT NULL,
  `name_last` varchar(25) COLLATE latin1_bin DEFAULT NULL,
  `email_address` varchar(255) COLLATE latin1_bin DEFAULT NULL,
  PRIMARY KEY (`human_id`),
  KEY `human_names` (`name_last`,`name_first`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 COLLATE=latin1_bin
```

这次，在那堆列的下面出现了一个新的 KEY。这个键，或者索引，叫 humans_names，它是根据括号中提到的两列的值来建立的。下面我们用 SHOW INDEX 来查看更多关于它的信息：

```
SHOW INDEX FROM birdwatchers.humans
WHERE Key_name = 'human_names' \G

*************************** 1. row ***************************
       Table: humans
  Non_unique: 1
    Key_name: human_names
Seq_in_index: 1
```

```
 Column_name: name_last
   Collation: A
 Cardinality: NULL
    Sub_part: NULL
      Packed: NULL
        Null: YES
  Index_type: BTREE
     Comment:
*************************** 2. row ***************************
       Table: humans
  Non_unique: 1
    Key_name: human_names
Seq_in_index: 2
 Column_name: name_first
   Collation: A
 Cardinality: NULL
    Sub_part: NULL
      Packed: NULL
        Null: YES
  Index_type: BTREE
     Comment:
```

上面这句展示了 human_names 索引的组成。结果有两行，行中有多列，它们展示这个索引是如何建立的。这里有很多关于这个索引的信息，但在这个阶段，你还没必要掌握它的全部含义。现在我只想让你看到，索引与它依赖的列的名字不同。即使索引只根据一列建立，而且索引名与列名相同，也不代表索引和列是同一个东西。

我们可以再跑一次 EXPLAIN...SELECT，看看跟之前没加索引的有什么区别：

```
EXPLAIN SELECT * FROM birdwatchers.humans
WHERE name_last = 'Hollar' \G

*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: humans
         type: ref
possible_keys: human_names
          key: human_names
      key_len: 28
          ref: const
         rows: 1
        Extra: Using where
```

如结果所示，这次 possible_keys 域显示可以用 human_names 键。如果可能用到的键不止一个，那么它们都会在这里列出。而在 key 域，我们看到 human_names 确实被用上了。基本上，只要用户按姓来查找，那么 MySQL 都会使用 human_names，而不会在表中逐行搜索。这也正符合我们的目的——使查询变得更快。

既然现在你已经更好地理解了索引以及它与列的关系，那么我们就回到之前的任务，将 conservation_status 表的 status_id 列改名为 conservation_status_id。因为索引与列关联，所以我们需先将这个关联去掉。否则，索引就会与一个不存在的列关联：因为它记住

的仍然是旧的列名。于是，我们先删掉索引，再改列名，最后基于新的列名来建立索引。即在 mysql 中输入以下 SQL 语句：

```
ALTER TABLE conservation_status
DROP PRIMARY KEY,
CHANGE status_id conservation_status_id INT PRIMARY KEY AUTO_INCREMENT;
```

注意这些子句的顺序必须如上所示，因为索引与该列关联，该列要在索引删除后才能重命名。你无需担心丢失数据：列中的数据并没有删除，只有索引被删除了，而紧接着，MySQL 会很轻易地将它重建。删除 PRIMARY KEY 时，不需要给出它所关联的列名。主键有且只有一个。

至此，你应该对索引，以及使用 ALTER TABLE 修改索引的过程有了更深入的认识。修改索引和它所依赖的列，是有先后次序的。其中的道理，你现在应该清楚了。不过，在本章结尾的习题中，我还是要考考你，我会让你再修改一些列和索引，以帮助你熟习这些概念和语法。请记得完成所有练习。

5.5 小结

好的规划对于数据库开发固然重要。然而，在以上那些使用 ALTER TABLE 的例子里，我们也看到了，MySQL 具备了足够的可塑性，它的数据库和表都能轻易地更改。你只需注意在重构前做好数据备份，并且在副本上进行改动。最后，检查清楚你的作业和数据，再提交结果。

记住了这些知识，以及尝试过本章的改表例子之后，你应该对建表感到毫无压力，因为你已经知道了没有必要在一开始就建得很完美。你还应该十分清楚列的各种选项，以及如何设置它们。此外，你还应基本了解了索引，以及如何创建、修改和使用它们。

如果对本章还有疑问，可能你还需要多接触一些带有数据的表。在本书的下一部分，你会有大量机会与表打交道，包括插入数据和修改数据。当数据来到你面前时，你会更加懂得如何为数据准备好表和列。你还会更好地认识到如何进行多表连接，以得出你想要的结果。

5.6 习题

除了你在本章输入过的 SQL 语句，这里还给出了一些习题，来巩固你刚刚学到的知识。这些题都跟建表和改表相关。最终产生的表会在后面的章节用到。所以，一定要完成这些练习。

(1) 在本章的前面，我们建了一个叫 birds_details 的表。该表有两列：bird_id 和 description。它们是从 birds 表中拿出来的。建立此表的目的是增加列来存放各种鸟的描述，记录它们的迁徙习性、生活区域，以及提供关于如何在野外识别它们的有用信息。现在我们来增加些列，以保存这些信息。

使用 ALTER TABLE 来修改此表。要求只写一条 SQL 语句，它能增加两列，分别叫

migrate 和 bird_feeder，都是整数类型（INT）。它们用于存放 1 或 0（即是或否）。另外，在这同一条语句中，还要使用 CHANGR COLUMN 子句，将 description 改名为 bird_description。

修改完后，用 SHOW CREATE TABLE 看看结果。

(2) 用 CREATE TABLE 语句创建一个叫 habitat_codes 的参考表。该表有两列：第一列是 habit_id，它是主键，需要 AUTO_INCREMENT，类型是 INT；第二列是 habitat，类型是 VARCHAR(25)。然后用以下语句插入数据：

```
INSERT INTO habitat_codes (habitat)
VALUES('Coasts'), ('Deserts'), ('Forests'),
('Grasslands'), ('Lakes, Rivers, Ponds'),
('Marshes, Swamps'), ('Mountains'), ('Oceans'),
('Urban');
```

执行一个 SELECT 语句来确认数据正常插入。该结果应如下：

```
+------------+----------------------+
| habitat_id | habitat              |
+------------+----------------------+
|          1 | Coasts               |
|          2 | Deserts              |
|          3 | Forests              |
|          4 | Grasslands           |
|          5 | Lakes, Rivers, Ponds |
|          6 | Marshes, Swamps      |
|          7 | Mountains            |
|          8 | Oceans               |
|          9 | Urban                |
+------------+----------------------+
```

再创建一个 bird_habitats 表。第一列叫 bird_id，第二列叫 habitat_id。两列类型都为 INT。两列都不需要索引。

建好这两个表后，用 DESCRIBE 和 SHOW CREATE TABLE 来展示它们。注意展示的结果，认清这两个表的结构和各列的组成。

使用 RENAME TABLE 将 bird_habitats 重命名为 birds_habitats。这个命令在 5.3.4 节中讲过。

(3) 用 ALTER TABLE 来给 bird_id 和 habitat_id 建个组合索引（5.4 节中讲过）。我们不用 INDEX 关键字，而用 UNIQUE，使其不重复。这个索引名为 birds_habitats。

改完后用 SHOW CREATE TABLE 查看。

到此，是时候给 birds_habitats 表添加一些数据了。用以下语句，看看 birds 和 habit_codes 都有些什么数据：

```
SELECT bird_id, common_name
FROM birds;

SELECT * FROM habitat_codes;
```

第一个语句的结果应该含有 loon 和 duck，以及其他鸟。loon 和 duck 都是可以在湖中找到的，其中 duck 还可以在沼泽中找到。所以，birds_habitats 表中需要有一行 loon 和

两行 duck。loon 的那行，用 loon 的 `bird_id` 和 Lakes, Rivers, Ponds 的 `habitat_id`。然后，duck 的一行，是 duck 的 `bird_id` 和 Lakes, Rivers, Ponds 的 `habitat_id`。接着，再来一行 duck，但 `habitat_id` 用 Marshes, Swamps 的。如果你建的索引正确，那应该是没有报错说记录重复的。做完之后，`SELECT` 出结果来看看。

(4) 用 `ALTER TABLE` 来给上一题的 `birds_habitats` 索引改名（这在本章临近尾声的地方介绍过）。将其改名为 `bird_habitat`。

(5) 再次使用 `ALTER TABLE`，给 `birdwatchers` 数据库的 `humans` 表增加三列，并且用同一语句完成。一列是 `country_id`，用于存放两个字符的国家代码，以示每个会员的所在地。接着是 `membership_type` 列，用于存放枚举值 `basic` 和 `premium`。第三列是 `membership_expiration`，类型是 `DATE`，以便我们得知 premium 用户何时过期。premium 用户在网站中拥有特权，而且在购买观鸟用品时享有优惠。

第三部分

数据处理基础

数据库的主要用途就是处理数据。在第二部分中，你已经学习了如何建表和改表。而数据处理方面的知识，也跟之前的内容一样，是有趣并且必要的。如果你在之前章节的建表和改表操作中感到疑惑，那可能是因为你缺乏插入数据的经验，所以难以想象表和列与数据的关系。

在这一部分中，我们将会探索几种将数据插入数据库和表的基本方法。这些会在第 6 章介绍。插入数据主要使用 `INSERT` 语句。而从表中获取数据则使用 `SELECT` 语句，这会在第 7 章详细介绍。在之前的章节中，我们其实已经使用它们好多次了。不过，在接下来的两章中，你会学到更多，包括它们的各种语法和选项。另外，我还为你准备了大量的练习。

除此之外，数据还经常需要更改甚至删除，所以，在第 8 章中，我们会介绍如何更新和删除数据。该章会讲述怎样使用 `UPDATE` 和 `DELETE` 语句，来完成这些常见的工作。它们对于数据管理是十分重要的。

而本部分的最后一章，即第 9 章，讲述的是更高级的问题。它虽然并不算很难，但你也不应该随便对待。具体来说，就是介绍如何从一个或多个表中选取数据，并以这些数据为基础，在其他表中进行插入、选取、更新或删除操作。因此，在读第 9 章之前，你应确保自己已经掌握了前面章节的基础知识。

本部分的各章都有一些示例代码，用于解释各种 SQL 语句及相关参数。这些你都应该输入一遍。即使你看的书是电子版，我也强烈建议你手工输入。这看上去好像是一件小事，但其实它能帮助你学习和记忆各种 SQL 语句的语法和差异。当你用错命令或打错字时，会得到报错信息。而分析这些报错信息，也是进步的途径。如果你只是将我呈现给你的东西复制粘贴进命令行，那么你就只是在给本书的示例勘误，这是学不到多少东西的。如果在学习过程中没有遇到挫折，是比较轻松的。而手敲代码虽然比较辛苦，但敢于犯错并分析问题，才能让你学到更多。

就跟本书的大多数章节一样，本部分的每一章结尾都有习题。就像我建议你敲打示例代码一样，你同样应该完成这些练习。本书不只是用于阅读，它还是你学习 MySQL 和 MariaDB 的工具。为了实现这个学习目标，你不应仅仅只是读书，还应参与、实验和钻研。如果你真的这么做，那么将会从本书获益良多。这些课后习题可能是本书最关键的部分，所以你应努力地完成它们。

第 6 章

插入数据

建立了数据库和表之后，下一步就是插入数据了。这里使用插入这个词，是因为往表中输入数据的最常用、最基本的方式，就是使用 INSERT 这条 SQL 语句。用关键字来指代所做的事情，能让你更轻松地学习 MySQL 和 MariaDB 的语言。本章将讨论 INSERT 的各种语法和选项，其中会用到在第 4 章创建过的表，以及在第 5 章更改过的表。除此之外，我们还会研究一些获取或选择数据的相关命令，这些命令会在第 7 章进行更详细的介绍。

在阅读本章的过程中，你还应该动手操作。当看到有 INSERT 或其他语句的示例代码时，你都应该试着用 mysql 客户端输入它们。而本章结尾的习题，你也应该做做。在做题过程中，你可能需要回顾本章和第 4 章中的一些示例。这能加深你对所学知识的印象。做完之后，你就能够轻松自如地在 MySQL 和 MariaDB 里插入数据了。

6.1 语法

INSERT 语句可向表中添加多行数据，可以一次一行，也可以一次多行。它的基本语法如下：

```
INSERT INTO table [(column, ...)]
  VALUES (value, ...), (...), ...;
```

关键字 INSERT INTO 后接一个表名，然后是括号中可选的列的集合。(方括号意味着其中的内容是可选的)。再接着是关键字 VALUES，以及用括号包围的对应各列的值的集合。INSERT 还有其他不同的语法，而现在说的这种是最基本的。还有，逗号是用于分隔的，如区分列名或各列的值。

现在我们用一些例子来演示 INSERT 的几个更简单的语法。你不用输入它们，因为这些例子用到的表都是我们没创建过的。

以下这个例子所涉及的语法是最简单的：

```
INSERT INTO books
VALUES('The Big Sleep', 'Raymond Chandler', '1934');
```

此命令向 books 表中输入了一些数据。这个表刚好只有三列，所以我们不用指定是哪三列。又或者说，因为没指定列，所以我们必须按 CREATE TABLE 时列的顺序，来提供这三列的值。于是，在这个例子里，The Big Sleep、Raymond Chandler 和 1934 会分别插入第一、二、三列中。

对于那些有设置默认值的列，你可以忽略该列的值，让服务器来帮你输入。一种做法是，在该列的位置上使用 DEFAULT 或 NULL，如下：

```
INSERT INTO books
VALUES('The Thirty-Nine Steps', 'John Buchan', DEFAULT);
```

这样 MySQL 就会给第三列插入默认值。如果默认值是 NULL——建表没指定默认值时就是 NULL——那么它就会给该行的该列插入 NULL。而如果用 AUTO_INCREMENT 指定了一列，那么服务器就会取该列的序列的下一个数来插入。

使用默认值的另一种方法是，指出不使用默认值的列，如下：

```
INSERT INTO books
(author, title)
VALUES('Evelyn Waugh','Brideshead Revisited');
```

注意此例我们用括号来指定了两列，而且顺序写反了。那么，在给值时，也要符合这个顺序。而第三列（即 year），就使用到默认值了。

如果你有多行需要插入，那么把它们都写在一条语句中会比较高效。这种语法跟之前的稍微不同。你需写多几对包含值集合的括号，并用逗号把它们分隔开，如下：

```
INSERT INTO books
(title, author, year)
VALUES('Visitation of Spirits','Randall Kenan','1989'),
      ('Heart of Darkness','Joseph Conrad','1902'),
      ('The Idiot','Fyodor Dostoevsky','1871');
```

这就输入三行数据到 books 表了。注意，我们只需指定列一次，VALUES 关键字也只写了一次。虽然子句后可跟多个项目或者一堆列表，但却不允许出现重复的子句（如本例的 VALUES），这是几乎所有的 SQL 语句都遵守的规则。

6.2 实例

现在，回到第 4 章和第 5 章操作过的 rookery 数据库，进行更多插入数据的实验。如果你没有创建表，那么请回过头去创建好再来学习本章。

通常人们都喜欢先往主表中添加数据，然后再往辅表或参考表添加数据，因为主表的内容比参考表有趣得多（但总的来说，数据库都是很无趣的，这是无法避免的）。虽然先插入主表也没问题，但这有可能造成数据冗余。

尽管我们都想避免数据冗余，但我还是建议，想要输入数据时再建表；输入完主表后，才输入辅表。这是因为我们很难一开始就确定到底需要多少表。相反，数据库开发是一个过程，你会不断地增加表，改表结构，甚至拆分表以改善性能、简化管理。而且也正是这样，才使得数据库没那么乏味，并令其变得好玩。

明白这个道理之后，在以后输入数据时，我们会先做一些基本判断，以决定需要什么表，先输入哪些表。先回忆一下我们是如何对鸟进行分类的：鸟种属于鸟科，鸟科属于鸟目。`birds` 表需要用 `family_id` 与 `bird_families` 表连接，`bird_families` 表需要用 `order_id` 与 `bird_orders` 表连接。所以，我们按以下顺序来输入数据：`bird_orders`、`bird_families` 和 `birds`。

考虑到大多数人都搞不清鸟种、鸟科和鸟目的学名，所以我会提供一些例子，让你能先插入几行测试数据（当然你也可以到维基百科和专门的观鸟网站或鸟类学网站，去查找这方面的信息，但学习本书不需要搞得这么复杂）。想要完整的数据，则可从我的网站（http://mysqlresources.com/files）下载。

6.2.1 鸟目表

在输入数据到 `bird_orders` 表之前，让我们通过执行以下 SQL 语句，来回忆一下该表的结构：

```
DESCRIBE bird_orders;

+-------------------+--------------+------+-----+---------+----------------+
| Field             | Type         | Null | Key | Default | Extra          |
+-------------------+--------------+------+-----+---------+----------------+
| order_id          | int(11)      | NO   | PRI | NULL    | auto_increment |
| scientific_name   | varchar(255) | YES  | UNI | NULL    |                |
| brief_description | varchar(255) | YES  |     | NULL    |                |
| order_image       | blob         | YES  |     | NULL    |                |
+-------------------+--------------+------+-----+---------+----------------+
```

如你所见，此表只有四列：一列是与 `bird_families` 表关联的标识号，一列是鸟目的学名，一列是鸟目的描述，还有一列是该鸟目的示例图。其中的 `order_id`，除非我们另外指定，否则会从 1 开始，并为以后插入的鸟目往上递增。

在输入鸟目数据之前，让我们先将 `order_id` 设为从 100 开始起跳，这样得到的标识号就至少是三位数了。该数字对 MySQL 是没什么意义的，这只是一种个人喜好。设置的方法，就是使用 `ALTER TABLE`（第 5 章讲过），如下：

```
ALTER TABLE bird_orders
AUTO_INCREMENT = 100;
```

这个命令虽说是更改 `bird_orders` 表，但实际修改的是服务器上保存 `AUTO_INCREMENT` 值的表。这会使得我们输入的第一个鸟目的 `order_id` 为 100。

接下来输入鸟目数据。我们可以用插入多行的语法来快速录入大批数据。因为现代鸟目也只有 29 种，所以我们可以一次就输入所有数据。下面这段长长的 SQL 语句就是我用来输数据的。你可以从我的网站下载这个表，又或者将以下 SQL 语句复制粘贴到 `mysql` 中：

```
INSERT INTO bird_orders (scientific_name, brief_description)
VALUES('Anseriformes', "Waterfowl"),
      ('Galliformes', "Fowl"),
      ('Charadriiformes', "Gulls, Button Quails, Plovers"),
      ('Gaviiformes', "Loons"),
      ('Podicipediformes', "Grebes"),
      ('Procellariiformes', "Albatrosses, Petrels"),
      ('Sphenisciformes', "Penguins"),
      ('Pelecaniformes', "Pelicans"),
      ('Phaethontiformes', "Tropicbirds"),
      ('Ciconiiformes', "Storks"),
      ('Cathartiformes', "New-World Vultures"),
      ('Phoenicopteriformes', "Flamingos"),
      ('Falconiformes', "Falcons, Eagles, Hawks"),
      ('Gruiformes', "Cranes"),
      ('Pteroclidiformes', "Sandgrouse"),
      ('Columbiformes', "Doves and Pigeons"),
      ('Psittaciformes', "Parrots"),
      ('Cuculiformes', "Cuckoos and Turacos"),
      ('Opisthocomiformes', "Hoatzin"),
      ('Strigiformes', "Owls"),
      ('Struthioniformes', "Ostriches, Emus, Kiwis"),
      ('Tinamiformes', "Tinamous"),
      ('Caprimulgiformes', "Nightjars"),
      ('Apodiformes', "Swifts and Hummingbirds"),
      ('Coraciiformes', "Kingfishers"),
      ('Piciformes', "Woodpeckers"),
      ('Trogoniformes', "Trogons"),
      ('Coliiformes', "Mousebirds"),
      ('Passeriformes', "Passerines");
```

在这么大段的 SQL 语句中，我只为每行指定了两列的值。order_id 是不需要指定的，因为我知道服务器会按我之前的要求，从 100 开始递增赋值。而 order_image 则会获得默认值 NULL，我们可以在以后想添加图像时再填写。另外，我们不能忽视已经指定的列。如果你的 INSERT 语句中没有为所有指定的列给出数据，那么 MySQL 将会拒绝执行它，并给你返回一个类似下面的报错信息：

```
ERROR 1136 (21S01):
Column count doesn't match value count at row 1
```

这意味着我们提供的值的数量与列的数量不匹配。

现在，你应该能看出为什么我要专门建个表来记录鸟目数据，而不是在鸟种表里为每种鸟都录入这些鸟目信息了吧。这是因为有了 bird_orders 表，你就可以在 bird_families 表里只填写 order_id。这就是参考表的好处之一。填写数字比填写学名简单，而且也降低了打错字的概率。

6.2.2　鸟科表

既然 bird_orders 表已经有数据了，那么就该轮到 bird_families 表了。首先，执行以下语句：

```
DESCRIBE bird_families;
```

它将显示 bird_families 的表结构。我们还需要知道鸟科所属鸟目的 order_id。现在，先从 Gaviidae 这个鸟科开始，它包括了 Great Northern Loon 这个鸟种——此种已在 birds 表里了。而它本身又属于 Gaviiformes 目。所以，我们先用以下 SQL 语句来查出该鸟目的 order_id：

```
SELECT order_id FROM bird_orders
WHERE scientific_name = 'Gaviiformes';

+----------+
| order_id |
+----------+
|      103 |
+----------+
```

接着，就可以这样把 Gaviidae 录入 bird_families 了：

```
INSERT INTO bird_families
VALUES(100, 'Gaviidae',
"Loons or divers are aquatic birds found mainly in the Northern Hemisphere.",
103);
```

此语句把 Gaviidae 的名字和描述输进了 bird_families。你可能已经发现，尽管我们为 family_id 设置了自动递增，但我却在这里指定了 100。这不是必要的，只是因为我喜欢标识号大于一位。如果 loons 这样优雅而又古老的鸟科被编号为 1，我会感觉不太好的。另外，因为指定了 100，这也使得服务器会为下一个鸟科带出 101 的标识号。

如果我们提供的值的数量对了，但顺序不合服务器的要求，那么服务器是不一定会接受的。假设我们想再给这个表增加一行——birds 表中 Wood Duck 所属的 Anatidae 科。而在写 SQL 语句时，我们给出的值的顺序与表结构的不同。这时，服务器还是会尽可能地执行它，只是结果不如我们所想。例如以下命令：

```
INSERT INTO bird_families
VALUES('Anatidae', "This family includes ducks, geese and swans.", NULL, 103);
Query OK, 1 row affected, 1 warning (0.05 sec)
```

注意此语句中我们把科的名字放在了第一位，然后是描述，再接着是 family_id NULL，最后是 order_id 103。但 MySQL 想要的第一列是数字，或 DEFAULT，或 NULL，而我们却给了它文本。所以，mysql 的返回信息说 Query OK, 1 row affected, 1 warning (0.05 sec)，意思是插入了一行，同时也产生了一条警告，只不过这个警告没显示出来。于是，我们用 SHOW WARNING 来查看它警告什么：

```
SHOW WARNINGS \G

*************************** 1. row ***************************
  Level: Warning
   Code: 1366
Message: Incorrect integer value: 'Anatidae' for column 'family_id' at row 1
1 row in set (0.15 sec)
```

这样我们就看到该警告了：family_id 列需要的是整数值，但得到的却是文本。那我们就用以下 SQL 语句来看看 bird_families 中录入数据后是什么样子：

```
SELECT * FROM bird_families \G

*************************** 1. row ***************************
        family_id: 100
  scientific_name: Gaviidae
brief_description: Loons or divers are aquatic birds
                   found mainly in the Northern Hemisphere.
         order_id: 103
*************************** 2. row ***************************
        family_id: 101
  scientific_name: This family includes ducks, geese and swans.
brief_description: NULL
         order_id: 103
```

第一行是对的，那是我们之前用正确的方法输入的。但第二行，因为 MySQL 没有获得像样的值，所以它就忽略了我们指定的值，而给该列赋了 101——这是根据 AUTO_INCREMENT 而得的。然后，它把我们想要赋予 brief_description 列的描述放到了 scientific_name 列，把想要赋予 family_id 的 NULL 放到了 brief_description。这样的结果是需要修改或删除的。我们就删掉它吧，用 DELETE：

```
DELETE FROM bird_families
WHERE family_id = 101;
```

这只会删掉一行：family_id 为 101 的那行。用 DELETE 时要小心。我们没有 UNDO 命令。如果你没加上 WHERE 子句，那就会删掉整个表的数据。对于这个只有两行数据的表来说，重新输入数据也不是什么问题。但如果是有数千行数据，而又没做备份，那么就可能永远都找不回来了。即使你有备份，恢复数据也不是能瞬间轻松做到的。所以，要小心使用 DELETE 语句，并且永远都加上 WHERE 子句来限定要删除的部分。

现在我们来重新输入 Anatidae，但这次换一种语法，让我们不用为所有列都指明值，也不用按表结构的顺序来列出这些值：

```
INSERT INTO bird_families
(scientific_name, order_id, brief_description)
VALUES('Anatidae', 103, "This family includes ducks, geese and swans.");
```

为了达到这个目的，我们在设置值之前，先用括号来指定要输入到哪些列中。如果所给的值能按顺序对上所有列，那么是没必要指定列的。但我们现在这条 SQL 语句不是这样，我们必须说明想要插入哪些列中，这些列怎样与 VALUES 子句里的值一一对应。简单来说，就是告诉服务器这些值代表什么，让服务器能把这些值放到对应的列里。而那些我们没给值的列，或没指定的列，服务器会给默认值。现在来看看插入的结果：

```
SELECT * FROM bird_families \G

*************************** 1. row ***************************
        family_id: 100
  scientific_name: Gaviidae
brief_description: Loons or divers are aquatic birds
```

```
                  found mainly in the Northern Hemisphere.
         order_id: 103
*************************** 2. row ***************************
        family_id: 102
  scientific_name: Anatidae
brief_description: This family includes ducks, geese and swans.
         order_id: 103
```

这就好多了。注意这次服务器按指令将科名 Antaidae 放到了 scientific_name 中。同时，它还给 family_id 赋予了一个数字值。因为此前设置过了 101（虽然后来被删掉），所以这样加 1 后，就把这行的标识号设为了 102。你可以更改此行的标识号并重置计数器（即此表的 family_id 的 AUTO_INCREAMENT 的值），不过其实也并不重要。

现在开始准备插入鸟科的数据。这次我们简单一点，只录入学名和鸟目标识号。首先，用以下 SQL 语句来查询各个鸟目的标识号：

```
SELECT order_id, scientific_name FROM bird_orders;

+----------+--------------------+
| order_id | scientific_name    |
+----------+--------------------+
|      100 | Anseriformes       |
|      101 | Galliformes        |
|      102 | Charadriiformes    |
|      103 | Gaviiformes        |
|      104 | Podicipediformes   |
|      105 | Procellariiformes  |
|      106 | Sphenisciformes    |
|      107 | Pelecaniformes     |
|      108 | Phaethontiformes   |
|      109 | Ciconiiformes      |
|      110 | Cathartiformes     |
|      111 | Phoenicopteriformes |
|      112 | Falconiformes      |
|      113 | Gruiformes         |
|      114 | Pteroclidiformes   |
|      115 | Columbiformes      |
|      116 | Psittaciformes     |
|      117 | Cuculiformes       |
|      118 | Opisthocomiformes  |
|      119 | Strigiformes       |
|      120 | Struthioniformes   |
|      121 | Tinamiformes       |
|      122 | Caprimulgiformes   |
|      123 | Apodiformes        |
|      124 | Coraciiformes      |
|      125 | Piciformes         |
|      126 | Trogoniformes      |
|      127 | Coliiformes        |
|      128 | Passeriformes      |
+----------+--------------------+
```

接着，使用一个巨大的 INSERT 语句来插入数据到 bird_families。其中每个科的数据都要用括号括起来，每对括号之间用逗号分隔。查询过观鸟指南后，我们知道了各科所属的

目，并写出了如下的 SQL 语句：

```
INSERT INTO bird_families
(scientific_name, order_id)
VALUES('Charadriidae', 109),
      ('Laridae', 102),
      ('Sternidae', 102),
      ('Caprimulgidae', 122),
      ('Sittidae', 128),
      ('Picidae', 125),
      ('Accipitridae', 112),
      ('Tyrannidae', 128),
      ('Formicariidae', 128),
      ('Laniidae', 128);
```

此语句一次插入了十行数据。注意我们不需要为每一行都指定列名。还有，我们这次没有提到 `family_id`。这样，服务器就会自动取该域的序列号的下一个值来填入其中。另外，我们也没有写 `brief_description` 列。这列会在以后我们想输入的时候再输入。

如果你希望这个表更大，并带有各科的描述，那么可以从我的网站上下载。暂时来说，这十行是足够的。现在执行一个 `SELECT` 语句，来看看有些什么 `family_id`。我们将会在录入 `birds` 表时用到它们。

```
SELECT family_id, scientific_name
FROM bird_families
ORDER BY scientific_name;

+-----------+-----------------+
| family_id | scientific_name |
+-----------+-----------------+
|       109 | Accipitridae    |
|       102 | Anatidae        |
|       106 | Caprimulgidae   |
|       103 | Charadriidae    |
|       111 | Formicariidae   |
|       100 | Gaviidae        |
|       112 | Laniidae        |
|       104 | Laridae         |
|       108 | Picidae         |
|       107 | Sittidae        |
|       105 | Sternidae       |
|       110 | Tyrannidae      |
+-----------+-----------------+
```

这里我还增加了一个 `ORDER BY` 子句，以确保结果是按照学名的字母序来显示的。关于 `ORDER BY`，我会在第 7 章详细介绍。

现在可以输入数据到 `birds` 表了。该表已有一个属于 Charadriidae 科的岸鸟 Killdeer。所以，我们给它增加几个同科的鸟。从上面的结果可看出，Killdeer 所属的 Charadriidae 的标识号是 103。不过，在你的机器上看到的可能和在我的机器上的不一样。

得知岸鸟的 `family_id` 以后，再看看 `birds` 表中我们要填哪些列。先用 `SHOW COLUMNS`：

```
SHOW COLUMNS FROM birds;

+-----------------------+-------------+------+-----+-------+----------------+
| Field                 | Type        | Null | Key |Default| Extra          |
+-----------------------+-------------+------+-----+-------+----------------+
| bird_id               | int(11)     | NO   | PRI | NULL  | auto_increment |
| scientific_name       | varchar(100)| YES  | UNI | NULL  |                |
| common_name           | varchar(255)| YES  |     | NULL  |                |
| family_id             | int(11)     | YES  |     | NULL  |                |
| conservation_status_id| int(11)     | YES  |     | NULL  |                |
| wing_id               | char(2)     | YES  |     | NULL  |                |
| body_id               | char(2)     | YES  |     | NULL  |                |
| bill_id               | char(2)     | YES  |     | NULL  |                |
| description           | text        | YES  |     | NULL  |                |
+-----------------------+-------------+------+-----+-------+----------------+
```

结果跟 DESCRIBE 没什么两样。不过，SHOW COLUMNS 语句可基于某些模式来过滤结果。例如，假设你只想查询用于参考的列——以 _id 结尾的列，那么你可以这么做：

```
SHOW COLUMNS FROM birds LIKE '%id';

+-----------------------+---------+------+-----+---------+----------------+
| Field                 | Type    | Null | Key | Default | Extra          |
+-----------------------+---------+------+-----+---------+----------------+
| bird_id               | int(11) | NO   | PRI | NULL    | auto_increment |
| family_id             | int(11) | YES  |     | NULL    |                |
| conservation_status_id| int(11) | YES  |     | NULL    |                |
| wing_id               | char(2) | YES  |     | NULL    |                |
| body_id               | char(2) | YES  |     | NULL    |                |
| bill_id               | char(2) | YES  |     | NULL    |                |
+-----------------------+---------+------+-----+---------+----------------+
```

我们使用了百分号（%）作为通配符——这里不是用星号的——以指定开头为任意字符，结尾为 _id 的这种模式。对于有大量列的表，这种过滤是挺有用的。所以，在起名字时，最好制定一个能方便日后查找的命名规范（例如，%_id）。顺便说一下，如果你在 SHOW COLUMNS 中加上 FULL 标识（如 SHOW FULL COLUMNS FROM birds;），那么就会看到各列更详细的情况。请在自己的系统上试试。

6.2.3 鸟种表

FULL 是挺好玩的，不过还是回到本章的重点——插入数据。回顾了 birds 表的结构以后，现在就输入一些岸鸟的数据吧。在 mysql 中执行以下命令：

```
INSERT INTO birds
(common_name, scientific_name, family_id)
VALUES('Mountain Plover', 'Charadrius montanus', 103);
```

这就添加了一个 Mountain Plover。注意，我打乱了列的顺序，但执行起来没有问题，因为后面值的顺序与这些列相符。这种鸟的 family_id 是 103，以示它属于 Charadriidae 科。这一行还有一些列没填数据，那些我们以后再弄。现在用 INSERT 的多行语法，再输入一些岸鸟：

```
INSERT INTO birds
(common_name, scientific_name, family_id)
VALUES('Snowy Plover', 'Charadrius alexandrinus', 103),
('Black-bellied Plover', 'Pluvialis squatarola', 103),
('Pacific Golden Plover', 'Pluvialis fulva', 103);
```

此例中，我们用一条语句插入了三个 family_id 相同的鸟种。这种做法我们之前在输入鸟科和鸟目时用过。注意这里用于 family_id 的数字没有引号包围。那是因为此列是整数类型的，即 INT，所以直接填数字是可以的。如果它们是带引号的，那么 MySQL 一开始会把它们当成字符，但经过分析，又会发现它们是数字，并将它们当数字存储。具体就是这样。或者简单来说，填数字列，带不带括号是没什么所谓的。

有了更多鸟的数据之后，我们就来把这些表连接起来查询数据吧。我们需要使用 SELECT 语句，并指定一系列的表以合并出结果。这比之前我们写过的任何一个 SELECT 都要复杂，它能让你看到各个不同的表的用途，尤其是那些参考表。试着在你的机器上输入以下 SQL 语句：

```
SELECT common_name AS 'Bird',
       birds.scientific_name AS 'Scientific Name',
       bird_families.scientific_name AS 'Family',
       bird_orders.scientific_name AS 'Order'
FROM birds,
     bird_families,
     bird_orders
WHERE birds.family_id = bird_families.family_id
AND bird_families.order_id = bird_orders.order_id;

+-----------------------+----------------------+--------------+---------------+
| Bird                  | Scientific Name      | Family       | Orders        |
+-----------------------+----------------------+--------------+---------------+
| Mountain Plover       | Charadrius montanus  | Charadriidae | Ciconiiformes |
| Snowy Plover          | Charadrius alex...   | Charadriidae | Ciconiiformes |
| Black-bellied Plover  | Pluvialis squatarola | Charadriidae | Ciconiiformes |
| Pacific Golden Plover | Pluvialis fulva      | Charadriidae | Ciconiiformes |
+-----------------------+----------------------+--------------+---------------+
```

我在此 SELECT 语句中连接了三个表。在看它查询哪些列之前，先来看看 FROM 子句里面有什么。注意其中我指明了三个表，并用逗号将它们分隔开来。为了让你看得明白，我还做了一些缩进。当然，表名是不一定要分行来写的。

MySQL 根据 WHERE 子句的内容来把这三个表串起来。首先，我们告诉 MySQL，把 birds 表与 bird_families 表中 family_id 相同的记录连接起来。然后，再用一个 AND，以指示后面还有其他条件。那就是把 bird_families 表与 bird_orders 表中 order_id 相同的记录连接起来。

这看起来挺复杂的。但如果你的手头上有一张包含数千种鸟的表格，以及一张鸟科表格和一张鸟目表格，并想用它们来列出每种鸟的名字、所属的科和目，那么应该也会做如同上例的事情——找出一张表中每一行的关键字，然后在其他表中以此关键字找出对应的行。这样想的话，就很直观了。

接着看看所查询的列。我们获取了来自 birds 表的 common_name 和 scientific_name，并且跟表名一样，我把列名也分行了，以方便查看。另外，因为三个表都有叫作 scientific_name 的列，所以我们必须在列名前加上表名以作区分。同时，我还用 AS 子句来给这些列起了别名，使结果的表头好看一点。AS 子句对服务器上的表没有任何影响，它只会作用于输出的结果。所以你可用它来任意指定结果中每列的标题。

现在来研究一下所得出的结果。每个科和目的学名都只需输入一次，然后 MySQL 就可以根据 family_id 和 order_id 把它们与多种鸟牵扯在一起。这实在是很酷、很省事。

如之前所说，这个 SQL 语句比以往的都要复杂。不过你也不用太担心。我们会在第 7 章讲解这种 SQL 语句。现在你只需明白我们这样做的目的是什么。这样把拆分的表连接起来查询，大大优于建立一个含有所有列的大表。对于每一种岸鸟，我们只需在 family_id 输入 103，而不用输入其所属科目的学名。这使得我们无需担心打错字，节约了时间，并有效利用了数据。

6.3 其他选择

本章之前提到过好几次，INSERT 语句还有别的写法。那么在本节中，我就来讲讲它们。刚开始可能你不会用到这些东西，但知道它们还是有必要的。

6.3.1 明确插入

INSERT 除了有基本的写法外，还有一种明确将值与列对应的写法。如下输入鸟科的例子。试试在 mysql 中输入，看喜不喜欢：

```
INSERT INTO bird_families
SET scientific_name = 'Rallidae',
order_id = 113;
```

它看起来有点奇怪。不过这种写法更能保证你不会输错，或至少能避免你搞乱列和值的顺序。因为它很死板，所以大多数人都不用它。但它所提供的准确性很适合于自动化脚本。因为它要求在值之前必须有列名，就像其他很多编程语言中都有的键值格式。这就方便了人们给脚本排错。另外，如果脚本写好后，列有改名或被删掉，那么该语句就会被服务器拒绝，数据不会被录入。不过，如果你用标准语法时写明列名了，那么其实也与此没什么区别。还有，这种语法每次只能插入一行数据。

6.3.2 插入其他表中的数据

INSERT 可与 SELECT 结合使用（第 5 章简单介绍过）。现在看看这有什么用。不过在此之前，我得先告诉你，本小节的示例有点复杂。你不一定要执行下面的示例，仅仅看看就好。

早前我们输入过 13 个鸟科。你可以通过下载我的网站上的鸟科表来录入所有鸟科，而我录入那个鸟科表时，其实是用的其他地方的数据（我总不能手工输入 228 行吧）。当时我去了康奈尔大学的网站。那里有个教授鸟类学的鸟类实验室，它是此学科的权威。在其网页上，有个公开的数据表。于是我将该表导入我的服务器上的 rookery 数据库，并将其命

名为 cornell_birds_families_orders。以下是该表的结构，以及其中数据的样子。

```
DESCRIBE cornell_birds_families_orders;

+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| fid         | int(11)      | NO   | PRI | NULL    | auto_increment |
| bird_family | varchar(255) | YES  |     | NULL    |                |
| examples    | varchar(255) | YES  |     | NULL    |                |
| bird_order  | varchar(255) | YES  |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+

SELECT * FROM cornell_birds_families_orders
LIMIT 1;

+-----+---------------+----------+------------------+
| fid | bird_family   | examples | bird_order       |
+-----+---------------+----------+------------------+
|   1 | Struthionidae | Ostrich  | Struthioniformes |
+-----+---------------+----------+------------------+
```

这些数据可以利用。我可以直接取科的名字，然后拿 examples 来作为描述，填充到 bird_families 表中。我不需要它们的标识号（即 fid），因为我要用自己的。另外我还需要一种能将此表中的 bird_order 列与我们 bird_orders 表的 scientific_name 列匹配的方法，以便找出正确的 order_id 填到 bird_families 表中。

可选的方法有很多。其中一种是，先增加一个列，以保存来自康奈尔的 bird_order。这要用到第 5 章介绍的 ALTER TABLE 语句，如下：

```
ALTER TABLE bird_families
ADD COLUMN cornell_bird_order VARCHAR(255);
```

加完以后，就可以用以下 SQL 语句，把 cornell_birds_families_orders 的数据复制到我们的鸟科表了：

```
INSERT IGNORE INTO bird_families
(scientific_name, brief_description, cornell_bird_order)
SELECT bird_family, examples, bird_order
FROM cornell_birds_families_orders;
```

仔细看一下这个命令，其中包含的知识你可能以后会用到。它首先如常地以 INSERT 开头，接着，在本应使用 VALUES 的地方，我却放置了一个完整的 SELECT 语句。这个 SELECT 的写法与我们之前所见的没有差别。它看上去很普通，但实际上很灵活而且很强大。

从概念上来说，你可以把它想象成，这个嵌入 INSERT 语句的 SELECT 产生了多行数据，每一行的值的顺序就如在 SELECT 中你指明的列的顺序一样。这些值跟 VALUES 子句类似，把给出的值送到父语句 INSERT，然后按顺序与列对应起来，填入其中。

此 INSERT 的开头有点不同，那就是增加了一个 IGNORE 选项。这是因为 bird_families 表已经存了一些数据，而 scientific_name 列设置为了 UNIQUE，不允许出现重复的学名。所以，如果这个多行插入因为产生了重复列名而出错，那么整个语句都会失败，并返回一个错误

信息。而 IGNORE 标志则指示服务器忽略所有错误，并插入那些没有产生错误的行。这就避免了失败和报错，你可以以后再查看那些另存起来的警告信息。这意味着，当服务器执行完这语句时，如果你想看报错的话，可以使用 SHOW WARNINGS 语句，来看看哪些行没有插入。这对于想要忽略重复而只处理非重复数据的人来说很有用。

既然录完数据了，那就在 mysql 中用以下 SQL 语句，来查询该表的最后一行——只显示一行就够了。

```
SELECT * FROM bird_families
ORDER BY family_id DESC LIMIT 1;

+-----------+-----------------+----------------+----------+-------------------+
| family_id | scientific_name |brief_description| order_id | cornell_bird_order|
+-----------+-----------------+----------------+----------+-------------------+
|       330 | Viduidae        | Indigobirds    |     NULL | Passeriformes     |
+-----------+-----------------+----------------+----------+-------------------+
```

这个 SELECT 语句是带有 ORDER BY 子句的，它使得查询结果按 family_id 来排序。而其后的 DESC，意思是逆序。LIMIT 子句则告诉 MySQL 只显示一行。从返回的结果可以看出，INSERT INTO...SELECT 运作正常。

6.3.3 题外话：设置正确的order_id

上例的 INSERT 使得我们从一个免费的数据库中获取到了所需的数据，并填入了我们自己的表中，但里面还差了一些数据：每个科所属的目的标识号。我们的目是定义在 bird_orders 表中的，每个目都有一个随意的 order_id。这些 order_id 与康奈尔的是不同的。所以我要根据 cornell_bird_order 列的值，查出 bird_orders 表对应的 order_id，来填到 bird_families 表的 order_id 中。

这看上去有些复杂，不过这个过程也展示了关系型数据库的强大。简单来说，我要用 bird_orders 表与康奈尔的数据做连接。之前我们已经从康奈尔那里得到了鸟目的名字。它们与我们 bird_orders 表里的 scientific_name 是一致的。但我们不想直接在 bird_families 表上标记鸟目的名字，而想用 order_id 来与 bird_orders 表关联。

要做好关联，就需要用 bird_orders 表的 order_id 列来设置 bird_families 表中的 order_id 列。要解决这个问题，得先找出 bird_orders 表中与 cornell_bird_order 列对应的行。

最终我们会用到 UPDATE 语句。不过在此之前，先构造一个 SELECT 语句来测试一下，以确保我们的鸟目名字能跟康奈尔的对得上。

```
SELECT DISTINCT bird_orders.order_id,
cornell_bird_order AS "Cornell's Order",
bird_orders.scientific_name AS 'My Order'
FROM bird_families, bird_orders
WHERE bird_families.order_id IS NULL
AND cornell_bird_order = bird_orders.scientific_name
LIMIT 5;

+----------+-----------------+-----------------+
| order_id | Cornell's Order  | My Order        |
```

```
+----------+-----------------+-----------------+
|      120 | Struthioniformes | Struthioniformes |
|      121 | Tinamiformes     | Tinamiformes     |
|      100 | Anseriformes     | Anseriformes     |
|      101 | Galliformes      | Galliformes      |
|      104 | Podicipediformes | Podicipediformes |
+----------+-----------------+-----------------+
```

这里测试的 WHERE 子句是我们接下来要在 UPDATE 中使用的。在实际 UPDATE 之前，最好先看看这个 WHERE 行不行。

此 WHERE 含有两个条件。第一个用于限制只更新那些未设 order_id 的行。这是合理的。如果已设好值，那没有理由再去改它。

而 AND 后就是第二个条件，它更重要。它根据康奈尔的学名找出对应的 bird_orders 记录，即查出与 cornell_bird_order 相等的 bird_orders 表的 scientific_name 所在的行。

如果你想用 INSERT…SELECT、REPLACE 或 UPDATE 来更改数据，那么可以先按上例来测试它们所带的 WHERE 子句。如果测试返回了所需的行，并且内容看起来没什么问题，那么就可以将该 WHERE 应用到其他更改数据的命令中了。

刚才这个 SELECT，与我们之前用于对 birds、bird_families 和 bird_orders 表进行关联查询的 SELECT 语句很像。只不过，它还增加了一个 DISTINCT 选项。这会只查出所有列都不同的行。因为 Struthioniformes 目包含的科超过五个，而我又限制了只输出五行（用 LIMIT 5），所以如果不加 DISTINCT 的话，就会看到第一行重复了五次。而加上 DISTINCT 就使得这五行都是不同的排列，也使我们更加相信这个 WHERE 是正确的。

因为查询结果无误，所以接下来就可以给 bird_families 表做 UPDATE 了。UPDATE 语句是用来修改或更新数据的。其基本语法是，指出你要更新的表的名字，然后用 SET 子句来设置各列的值。它有点像 6.3.1 节中的 SELECT 语句。你还可以使用 WHERE 来指定要更新哪些行：

```
UPDATE bird_families, bird_orders
SET bird_families.order_id = bird_orders.order_id
WHERE bird_families.order_id IS NULL
AND cornell_bird_order = bird_orders.scientific_name;
```

以上语句相当复杂，让我们再逐步分析一下此 UPDATE 语句干了什么：它告诉 MySQL，将 bird_families 表的 order_id 设置为 bird_orders 表中对应行的 order_id 的值——但因为后面的 AND 子句，它只会作用于 cornell_bird_order 与 bird_orders 表中 scientific_name 相等的行。

我知道这涉及了很多东西。我会在第 8 章更详尽地介绍这条语句。

现在来看看更新的结果。我们会用之前用过的那个 SELECT，但这次 LIMIT 为 4，以看多几行：

```
SELECT * FROM bird_families
ORDER BY family_id DESC LIMIT 4;

+-----------+-----------------+-------------------+----------+
| family_id | scientific_name | brief_description   | order_id |
```

```
+-----------+----------------+--------------------+----------+
|       330 | Viduidae       | Indigobirds        |      128 |
|       329 | Estrildidae    | Waxbills and Allies |     128 |
|       328 | Ploceidae      | Weavers and Allies |      128 |
|       327 | Passeridae     | Old World Sparrows |      128 |
+-----------+----------------+--------------------+----------+
```

这看上去好像成功了。Viduidae 科的 `order_id` 列有值了，不再是 NULL。那就再检查一下 `bird_orders`，看看这个值正不正确：

```
SELECT * FROM bird_orders
WHERE order_id = 128;

+----------+-----------------+-------------------+-------------+
| order_id | scientific_name | brief_description | order_image |
+----------+-----------------+-------------------+-------------+
|      128 | Passeriformes   | Passerines        | NULL        |
+----------+-----------------+-------------------+-------------+
```

这是正确的。128 这个 `order_id` 是 Passeriformes，康奈尔也说它是 Viduidae 科所属的目。接着，看看 `bird_families` 中哪些行是找不到 `order_id` 的：

```
SELECT family_id, scientific_name, brief_description
FROM bird_families
WHERE order_id IS NULL;

+-----------+------------------+----------------------+
| family_id | scientific_name  | brief_description    |
+-----------+------------------+----------------------+
|       136 | Fregatidae       | Frigatebirds         |
|       137 | Sulidae          | Boobies and Gannets  |
|       138 | Phalacrocoracidae | Cormorants and Shags |
|       139 | Anhingidae       | Anhingas             |
|       145 | Cathartidae      | New World Vultures   |
|       146 | Sagittariidae    | Secretary-bird       |
|       147 | Pandionidae      | Osprey               |
|       148 | Otididae         | Bustards             |
|       149 | Mesitornithidae  | Mesites              |
|       150 | Rhynochetidae    | Kagu                 |
|       151 | Eurypygidae      | Sunbittern           |
|       172 | Pteroclidae      | Sandgrouse           |
|       199 | Bucconidae       | Puffbirds            |
|       200 | Galbulidae       | Jacamars             |
|       207 | Cariamidae       | Seriemas             |
+-----------+------------------+----------------------+
```

因为某些原因，`bird_orders` 表中未能找到与这 15 行匹配的数据。那么我得查查为什么。下面来看看我是怎么解决的。

我先查了一下 Osprey，发现它有两个可用的目名：Accipitriformes 和 Falconiformes。其中 Accipitriformes 是康奈尔的叫法，而 Falconiformes 是我的 `bird_orders` 表的叫法（在 `order_id` 为 112 的那行）。我决定用 112 来更新 `bird_families` 表：

```
UPDATE bird_families
SET order_id = 112
WHERE cornell_bird_order = 'Accipitriformes';
```

我本来可以在 WHERE 子句中用 family_id 的，但我用 Accipitriformes 一查，发现还有另外两个科，所以用 cornell_bird_order = 'Accipitriformes' 可以一次更新这三行。再检查一下，我还发现有四个鸟科属于一个新的叫 Suliformes 的鸟目。于是我就在 bird_orders 表中加上了这个 Suliformes，然后更新了一下那四个科的 order_id。在创建数据库或从其他库导入大量数据时，这种清理方法是很常见的。

那么，剩下的清理工作就是：删掉 bird_families 表中那个额外的列（即 cornell_bird_order 列），以及 cornell_birds_families_orders 表：

```
ALTER TABLE bird_families
DROP COLUMN cornell_bird_order;

DROP TABLE cornell_birds_families_orders;
```

以上这些例子是有点难，所以如果你感到困惑，也不要气馁。很快，你就能亲手构造出更复杂的 SQL 语句了。而且，到时你回过头来看，会发现其实这些例子还可以简化。另外，我还想告诉你，强大的不止是 MySQL 和 MariaDB，还有它们的社区。之所以提及社区，是因为在那里，你可以找到一些免费下载的数据表，以便自己把玩。这就让你省了很多数据库管理的工作，玩起来没那么闷。导入大批数据的方法还有很多，甚至还可以导入一些不是来自 MySQL 表的数据。具体会在第 15 章介绍。

6.3.4 替换数据

当你使用多行语法来给一个现存的表插入数据时，如果其中有操作到声明为键的列，那就可能会出现一些问题。如上例中的 bird_families 表，它的 scientific_name 是一个 UNIQUE 键，即每个科的学名只允许有一条记录。如果 MySQL 在执行 INSERT 语句时发现有键重复，那么它就会拒绝执行，并返回一个报错信息。最后什么都不会插入。

于是你得改改那个可能很长的 INSERT 语句，剔除掉重复项后，再运行一次。如果重复项很多，那么就会多次接收错误信息，然后多次修改，直至插入成功。为了避免这样，在上面的例子中，我们用了 IGNORE 选项。它告知 MySQL 忽略错误，重复的就不插入，不重复的才插入。

不过很多时候，你可能不想略过那些重复的行，而想在发生重复时，用新的行来替换旧的行。比如在上例，因为我们有更新更好的信息，所以就使用了 UPDATE 来覆盖掉旧的那些。对于这种情况，你可以使用 REPLACE 语句，而不是 INSERT。REPLACE 跟 INSERT 一样，也是插入数据。但在遇到键重复时（例如重复的 scientific_name），它会采取替换的做法。这很有用，而且也不难。现在看看例子：

```
REPLACE INTO bird_families
(scientific_name, brief_description, order_id)
VALUES('Viduidae', 'Indigobirds & Whydahs', 128),
('Estrildidae', 'Waxbills, Weaver Finches, & Allies', 128),
('Ploceidae', 'Weavers, Malimbe, & Bishops', 128);
```

```
Query OK, 6 rows affected (0.39 sec)
Records: 3  Duplicates: 3  Warnings: 0
```

注意它的语法跟 INSERT 一样，选项也一样，也同样可以一次插入多行。不过，IGNORE 选项是不需要的，因为重复的行已被替换了。

事实上，REPLACE 所做的替换，是先把原本的整行删除掉，然后再插入新的那行。如果新行中有些列没有值，那么就会使用默认值。它不会保留原本行的任何列。如果有些列是需要保留的，那么使用它就要小心了，因为你无法在 REPLACE 中选择只更新某些列。要更新指定的列，请用 UPDATE。

以上 REPLACE 及其所带的值，还有一些值得注意的东西。在结果中，你可以看到一些不寻常的信息。它说此 SQL 语句影响了六行数据：三行新增的和三行重复的。六行受影响，听起来很奇怪。那是因为原本有三行跟新增的三行在 scientific_name 上重复了，于是就被删除了。然后新增的三行，或者说用于替换的三行，被插入了。所以总共六行受影响：删了三行，又加了三行。

结果中没有任何警告信息，这意味着 MySQL 认为一切正常。现在看看其中一个被修改的科，Viduidae：

```
SELECT * FROM bird_families
WHERE scientific_name = 'Viduidae' \G

*************************** 1. row ***************************
        family_id: 331
  scientific_name: Viduidae
brief_description: Indigobirds & Whydahs
         order_id: 128
```

虽然不太明显，但确实所有东西都换掉了。family_id 是新的。如果你返回前面，会发现之前它是 330。因为当时它是此表的最后一行，所以在替换时，所插入的新行就被赋值为 331 了。而 brief_description 也出现新值了，以前它是只有 Indigobirds 的。

REPLACE 语句可用于替换含有重复键的整行数据，或新增原表中所没有的数据。如果你只想替换某些列，那用它就有点问题了，因为它是整行替换的。另外，如果上例中的 scientific_name 不是 UNIQUE 或键，那么 REPLACE 的结果就不是替换了三行，而是新增了三行。

6.3.5 数据插入的优先级

在一台繁忙的 MySQL 或 MariaDB 服务器上，可能经常会发生多人同时访问的情况。于是就会有来自不同客户端的 SQL 语句被同时输入。这使得服务器需要判断应该让哪个先执行。

更改数据的语句（INSERT、UPDATE 和 DELETE），会比查询语句（SELECT）具有更高的优先级。添加数据的人应该比读取数据的人更重要，因为考虑到插入数据可能占用较长时间，在这期间客户端做不了其他事情。而相反，查询数据的人一般都愿意等待。就像在购物网

站中，下单的用户会比浏览产品的用户，具有更高的优先级。

当服务器为客户端执行一个 INSERT 语句时，它会锁住相关的表，以排斥其他客户端的访问，直到它执行完毕。不过 InnoDB 倒不是这样：它只锁住行，而不是整个表。对于一个有大量并发数据请求的繁忙的服务器来说，锁表会导致其他用户延迟，尤其是在有人使用多行语法来插入大量数据的时候。

你可以为每个 INSERT 另外设定优先级，而不采用 MySQL 的默认设置。这样你就可以决定哪些语句需要立即执行，哪些可以缓后。INSERT 语句提供了优先级选项，以便你指定这些偏好。选项要写在 INSERT 和 INTO 之间。它们总共有三种：LOW_PRIORITY、DELAYED 和 HIGH_PRIORITY。下面就来看看每一个是怎么回事。

1. 调低INSERT的优先级

先举个 LOW_PRIORITY 的用例。假设我们刚从一个观鸟组织获取到了一个含有数千行观鸟景点相关数据的文件。它是一个 dump 文件，即一个包含了必要的数据导入语句的文本文件。用文本编辑器打开它，可看到里面有一个一次就插入所有景点（bird_sightings）的巨型 INSERT 语句。虽然实际上我们没有建这样的表，但你应该可以想象它的结构。

一旦这个 INSERT 语句运行起来，它应该会使得我们的服务器卡住一段时间。而现在我们希望，如果有人正在查询 bird_sightings 表的数据，那就把 INSERT 押后，等那人查完再执行。LOW_PRIORITY 就能做到这一点，它将等待 MySQL 做完其他所有事情后，再输入数据。以下是它的使用方法：

```
INSERT LOW_PRIORITY INTO bird_sightings
…
```

当然，现实中的 INSERT 语句会有列声明和值的集合（在省略号处）。

LOW_PRIORITY 标志使得该 INSERT 语句被放到一个队列中，并等待其他正在处理和准备处理的请求完成后，再去执行。如果低优先级的语句未被处理，而又有人发起一个新的请求，那么这个请求也是放在队列的前端。MySQL 只会在处理完前面的请求之后，才去执行低优先级的语句。

插入执行时，其他请求都得等待它做完才能被处理。低优先级的语句开始执行时，MySQL 就会锁住相关的表，以阻止其他并发的插入。MySQL 不会中断进行中的 LOW_PRIORITY 插入，并容许其他数据更改。顺便说一下，InnoDB 引擎的表不支持 LOW_PRIORITY 和 HIGH_PRIORITY，因为 InnoDB 只锁定相关的行而不锁定整个表，所以这两个选项对它没有意义。

因为 mysql 必须等待服务器执行完语句才能恢复使用，所以 INSERT LOW_PRIORITY 可能会造成一些不便。如果你用 mysql 连接一个繁忙的服务器，并执行低优先级的 INSERT，那么你的 mysql 就可能会定住几分钟，甚至几小时，这具体得看那个服务器忙到什么程度。LOW_PRIORITY 会使你客户端一直处于等待状态，直到服务器开始执行插入，然后客户端被锁定，就如同表被锁定一样。

2. 延迟插入

你可以不用 LOW_PRIORITY，而用 DELAYED。不过 MySQL 从 5.5.6 开始，不提倡使用它了。如果你还用着旧版本，那么可以这样用它：

```
INSERT DELAYED INTO bird_sightings
...
```

这跟 LOW_PRIORITY 很像，MySQL 会将此语句先记为低优先级，然后在空闲时再执行。它与 LOW_PRIORITY 相比，区别在于，或者说优势在于，它不会定住 mysql，所以你可以接着输入其他命令，甚至退出 mysql。另外，多个 INSERT DELAYED 还可以被集合起来，在服务器空档时成批执行，这样可能会比 INSERT LOW_PRIORITY 更加高效。

而缺点就是，它不会通知客户端延迟插入是否已做。如果该 SQL 语句解析得有错，那么你还是能收到错误信息的，因为 SQL 语句在入队前，必须先解析验证。但是它执行时所产生的问题，不会返回给你。

由此我们还能看出另一个问题：延迟的插入命令保存在服务器内存中。如果 MySQL 守护进程停掉，或被人为杀掉，那么该命令就会丢失，而客户端也不会收到失败提示。你只能自行检查数据或者查看服务器日志，来确定它是否插入成功。所以，DELAYED 也不总是绝佳的选择。

3. 提升INSERT的优先级

HIGH_PRIORITY 是第三种优先级选项。你可能会觉得它没有必要，因为 INSERT 语句本来默认就是优先于其他只读的 SQL 语句的。但是，这个默认设定（即 INSERT 先于 SELECT）是有可能被改写的。在 2.6 节中，我们提过 MySQL 和 MariaDB 的配置。其中一项就是 --low-priority-updates。它会使得写入语句变为低优先级语句，或使其优先级至多等同于只读语句。如果你的服务器被设定了这一项，那么你可以在 INSERT 中加上 HIGH_PRIORITY 以覆盖该设定，使 INSERT 先于 SELECT。

6.4 小结

现在，你应该对 MySQL 和 MariaDB 有了清晰的理解。你应该知道了数据库和表的基本结构，还应该认识到了构建多个小表的好处。你不会再将数据库看成一个巨大的表，或者是一个电子表格系统。你应该知道列是什么，以及如何往其中输入数据。如果你已做完前两章的习题，那么这对你来说应该是很简单的问题。暂时来说，掌握以上内容就够了。

第 7 章会深入讲解如何使用 SELECT 语句从表中获取数据。此前我们已多次用过 SELECT。然而，那都只是为了验证数据输入而做的，只使用到 SELECT 的一般功能。下一章会介绍 SELECT 的更多细节。

INSERT、SELECT 和 UPDATE 是最常使用的 SQL 语句。如果你想学好 MySQL 和 MariaDB，那就必须先掌握它们。关于 SELECT 语句，除了基本的操作，你还得熟悉它一些特别的使用方法。而读完下一章，你就能做到了。

不过，在进入下一章之前，先完成以下习题。它们能助你巩固本章所学的 INSERT 语句的知识。请不要跳过它们，因为它们能为你的学习打好基础。

6.5 习题

以下是有关使用 INSERT 语句和本章所学其他内容的一些习题。其中有些习题需要你先创建本章提到的一些表，所以你要做的不仅仅是插入数据。建表的练习能使你更好地理解数据的输入。而输入数据又能让你对建表有更加理智的认识。它们是相辅相成的。

(1) 第 4 章结尾曾经要求你建立一个 birds_body_shapes 表。此表的数据可用于识别各种鸟。它的 body_id 会被 birds 表引用。该表包含鸟的各种体型的描述，并且这个描述是识别鸟种的关键信息：如果有种鸟看起来像鸭子，走起来像鸭子，叫起来也像鸭子，那么它可能是鹅——但绝不可能是蜂鸟。以下就是体型名称的列表：

    Hummingbird
    Long-Legged Wader
    Marsh Hen
    Owl
    Perching Bird
    Perching Water Bird
    Pigeon
    Raptor
    Seabird
    Shore Bird
    Swallow
    Tree Clinging
    Waterfowl
    Woodland Fowl

    用多行语法（不是明确插入的那种），写一个 INSERT 语句，用于插入数据到 birds_body_shapes 表。其中 body_id 是三个字母组成的代码，具体内容由你来定，你也可以参考体型的名称（例如，MHN 代表 Marsh Hen，而 Owl 就直接用 OWL）。只要是各不相同就行。而 body_shape 列，就用上面给的内容填，当然你也可以自定名称。第三列 body_example，暂时先不管。

(2) 除此之外，第 4 章结尾还让你建 birds_wing_shapes 表，它用来存放鸟翼形状的数据，也是用于识别鸟种的。以下是那些形状的名称的初始列表：

    Broad
    Rounded
    Pointed
    Tapered
    Long
    Very Long

    用明确插入的方式（带 SET 子句），构造一个 INSERT 语句，将以上项目插入 birds_wing_shapes 表中。其中 wing_id 由两个字母组成。内容随你定，就像刚才的 body_id 那样。wing_shape 用上面给出的内容填。而 wing_example 暂时不输入。

(3) 最后一个鉴别鸟种的表是 birds_bill_shapes。也是用 INSERT 来录入数据，但语法随你喜欢。bill_id 使用你自定的两个字母。bill_example 不输入。bill_shape 使用以下列表：

All Purpose
Cone
Curved
Dagger
Hooked
Hooked Seabird
Needle
Spatulate
Specialized

(4) 用 SELECT 语句，查出 birds_body_shapes 表中 body_shape 为 Woodland Fowl 的行。然后把它 body_shape 的值改成 Upland Ground Birds。要求使用 6.3.4 节中提及的 REPLACE 语句。你可在其 VALUES 子句中，带上刚才查出的该行的 body_id，以使其真的执行替换动作。

执行完 REPLACE 后，用 SELECT 来查询 birds_body_shapes 表的所有行。检查那一行改成了什么样子。确保它改对了。如果不对，用 REPLACE 或 UPDATE 再做一次。

第 7 章

查询数据

前面几章讲了两个重要的问题：如何组织好表，以及如何将数据录入表中。但要是不懂怎样从数据库中获取数据，那你放多少东西进去都没什么用。所以，本章就来谈谈数据库查询。

要从 MySQL 或 MariaDB 获取数据（或者说查询数据），最简单的方法就是使用 SQL 语句 SELECT。之前的章节中我们有几次用到过它，而这一章将会对它进行详解。我们没有必要知道或使用它所有的形式，而像表连接这样的技术，则是使用关系型数据库的基本技能，是我们必须懂得的。

我们会先回顾一下有关 SELECT 语句的基础知识，然后再看看它的各种变体。学完本章之后，你会懂得如何使用 SELECT 语句完成数据库开发的大部分工作，并准备好迎接未来各种可能发生的特殊情况。

在之前的章节中，尤其是在习题中，我曾请你往那些我们创建和更改过的表中录入数据。当时的手动录入是一种练习。但现在我们需要更多的测试数据，以便本章的示例代码能得出一些比较贴近现实的结果。请到本书的网站（http://mysqlresources.com/files）下载含有大量数据表的 dump 文件。

请下载 rookery.sql，来获取整个 rookery 数据库，其中包括大量的数据，我们可以在学习的过程中使用它。下载好 dump 文件后（假设你把它放在 /tmp/rookery.sql），在命令行输入以下命令：

```
mysql --user='your_name' -p \
rookery < /tmp/rookery.sql
```

此命令会根据所填用户名来提示输入密码，然后登录，在 rookery 数据库里执行 rookery.sql 文件中的语句。若一切正常，应该不会有任何回应，最后会返回到命令行提示符。

7.1 基本查询

SELECT 语法的基本元素，就是 SELECT 这个关键字，还有你想查询的列，以及这些列所属的表：

```
SELECT column FROM table;
```

如果你想查询多列，就以逗号分隔。如果想查询所有列，可以用星号通配符，而不需要将所有列名全部写出。现在就用刚刚导入了数据的 rookery 数据库，来举个基本语法的例子。输入以下语句，以获取 birds 表的所有行和所有列：

```
USE rookery;

SELECT * FROM birds;
```

这是能成功执行的最短小的 SELECT 语句。它指示 MySQL 获取 birds 表内的所有数据。结果中列的顺序和你在 CREATE TABLE 或 ALTER TABLE 中所定义的一样，而行的顺序则按照它们在表中被找出的顺序，通常会如同插入时的顺序。

若只想查出某些列，可以用类似下面的语句：

```
SELECT bird_id, scientific_name, common_name
FROM birds;
```

此语句只查出 birds 表中每一行的三个列。当然，只查出某些行，按某种顺序显示行或限定显示的行数，都是可以的。这些会在本章接下来的几节中讲到。

7.2 有条件地查询

假设我们只想查出某一科的鸟，如 Charadriidae（即 Plover）。从 bird_families 表可知，它的 family_id 是 103。于是，我们在 SELECT 语句中使用 WHERE 子句，从 birds 表中获取这一科的鸟种列表：

```
SELECT common_name, scientific_name
FROM birds WHERE family_id = 103
LIMIT 3;

+----------------------+-------------------------+
| common_name          | scientific_name         |
+----------------------+-------------------------+
| Mountain Plover      | Charadrius montanus     |
| Snowy Plover         | Charadrius alexandrinus |
| Black-bellied Plover | Pluvialis squatarola    |
+----------------------+-------------------------+
```

此 SELECT 语句要求返回两列，并且排位与定义表时所排的“scientific_name 先于 common_name”不同。并且我还加了 LIMIT 子句，使结果只显示头三行。至于 LIMIT 子句，我会在稍后讲到。

因为我们将科的信息放在了另一个表中，所以必须在查询 birds 之前，先从 bird_families 找出相应的 ID。这看起来好像绕了一圈。其实我们有更简捷的方法，可以直接使用 Charadriidae 而不是一个数字来查询。该方法叫作表连接。我们会在后面讲到。

以上例子很简单，而且我们在之前的章节中也看到过。接下来我们再进一步，看看如何调整结果集的顺序。

7.3 结果排序

上例从 birds 表获取了指定的列，并且用 LIMIT 来限定了行数。所得的行的排序方式取决于它们在表中被找到的顺序。因为我们只显示 Charadriidae 科所含鸟种的一小部分，所以排序对最终结果有很大影响。如果想让结果按照 common_name 列的值的字母序来排列，可以这样加上 ORDER BY 子句：

```
SELECT common_name, scientific_name
FROM birds WHERE family_id = 103
ORDER BY common_name
LIMIT 3;

+----------------------+----------------------+
| common_name          | scientific_name      |
+----------------------+----------------------+
| Black-bellied Plover | Pluvialis squatarola |
| Mountain Plover      | Charadrius montanus  |
| Pacific Golden Plover | Pluvialis fulva     |
+----------------------+----------------------+
```

注意，ORDER BY 是放在 WHERE 之后、LIMIT 之前的。于是，此语句不但使结果按 common_name 来排序，还按此顺序取头三行。这意味着，MySQL 先按 WHERE 子句获取所有行，并在幕后将此结果集存放于一个临时表中，然后根据 ORDER BY 子句对该表排序，最后根据 LIMIT 子句获取排序后的表的前三行。这就是为什么我们要将这三个子句按这样的次序来写。

ORDER BY 子句默认是升序的，即从 A 到 Z 往下排。如果你想逆序，可加上 DESC 选项，即 ORDER BY DESC。相对的，也有表示升序的 ASC 选项，不过一般是不需要用的，因为默认就是升序。

若想按多列排序，只需在 ORDER BY 后给出这些列，并用逗号分隔即可。你可以指定各列升序还是逆序。这种写法会使得结果先按首先指定的列来排，之后再按后指定的列来排，并且不违反前面已排好的顺序。为了演示这种排序方式，我们再多查一列，即 family_id，并且不只查一科。我们会查出更多的岸鸟：Oystercatchers（即 Haematopodidae）、Stilts（即 Recurvirostridae）和 Sandpipers（即 Scolopacidae）。首先输入以下命令，得出这些科的 family_id：

```
SELECT * FROM bird_families
WHERE scientific_name
```

```
IN('Charadriidae','Haematopodidae','Recurvirostridae','Scolopacidae');

+-----------+------------------+-----------------------------+----------+
| family_id | scientific_name  | brief_description           | order_id |
+-----------+------------------+-----------------------------+----------+
|       103 | Charadriidae     | Plovers, Dotterels, Lapwings |      102 |
|       160 | Haematopodidae   | Oystercatchers              |      102 |
|       162 | Recurvirostridae | Stilts and Avocets          |      102 |
|       164 | Scolopacidae     | Sandpipers and Allies       |      102 |
+-----------+------------------+-----------------------------+----------+
```

我们在此语句的 WHERE 子句中增加了一个 IN 运算符。在其后的括号中，我们可以指定多个用于匹配 scientific_name 的值。现在，查询 birds 表，再一次使用 IN，并且加上 LIMIT 子句：

```
SELECT common_name, scientific_name, family_id
FROM birds
WHERE family_id IN(103, 160, 162, 164)
ORDER BY common_name
LIMIT 3;

+-------------+--------------------------------+-----------+
| common_name | scientific_name                | family_id |
+-------------+--------------------------------+-----------+
|             | Charadrius obscurus aquilonius |       103 |
|             | Numenius phaeopus phaeopus     |       164 |
|             | Tringa totanus eurhinus        |       164 |
+-------------+--------------------------------+-----------+
```

注意，之前查 bird_families 时，IN 中的值加了引号，而这次的数字值则没有加引号。对于字符串来说，我们必须要加单引号或双引号。而对于数字值来说，最好不要加引号，因为这会影响性能，而且跟字符串混在一起时，会导致错误的结果。

此结果集有个很奇怪的地方：全都没有俗名。这是正常的。birds 表中大约有 10 000 种鸟是真种，还有 20 000 种左右是亚种。很多亚种是没有自己的俗名的。真种和亚种的数量如此之多，而亚种之间的差别又如此之小，所以没办法给它们全部都起个俗名。鸟类学家会给每种鸟都起个学名，而普通人看不出各种鸟之间的细微差异，会把一个俗名用在多种鸟上。所以 scientific_name 是必填的，而 common_name 则不是，它不能作为键。

现在再执行一次刚才的 SQL 语句，但在 WHERE 子句中增加一个参数，限定只显示 common_name 有值的行：

```
SELECT common_name, scientific_name, family_id
FROM birds
WHERE family_id IN(103, 160, 162, 164)
AND common_name != ''
ORDER BY common_name
LIMIT 3;

+----------------------+----------------------+-----------+
| common_name          | scientific_name      | family_id |
+----------------------+----------------------+-----------+
| African Oystercatcher | Haematopus moquini   |       160 |
```

```
| African Snipe         | Gallinago nigripennis |       164 |
| Amami Woodcock        | Scolopax mira         |       164 |
+-----------------------+-----------------------+-----------+
```

在 WHERE 子句中，我们使用逻辑运算符 AND，添加了第二个过滤条件。这样就能查出 family_id 在 103、160、162、164 之中，而且 common_name 不为空的行。

非程序员可能需要先了解一些编写大型 WHERE 子句的惯例。之前我们见过，等号可用于表示“该列必须含有此值”，而现在看到的 != 则是指“该列不能含有此值”。在上例中，我们用了 '' 来指代空字符串。于是，我们就查出了具有俗名的行。

注意，我们并不是要求非 NULL。你可以让 birds 表的 common_name 默认为 NULL，但我还是决定用空字符串。它与 NULL 是完全不同的：NULL 表示没有值，而空字符串虽然不含字符，但它仍然是一个字符串。因为设置了默认值是空字符串，所以匹配时也得用相应的值。

顺便说一下，!= 还可以写成 <>（即一个小于号后接一个大于号）。

7.4　限定结果集

birds 表有将近 30 000 行，如果不限定查询的数据，那返回结果可能会多于你一次想看的数量。而 LIMIT 就是用于解决这个问题的，我们已经试过用它来将 SELECT 的结果限定为三行，当时是根据 WHERE 和 ORDER BY 所得出的结果显示头三行。如果想查看接下来的行，比如说根据之前条件所得的结果集的下两行，那么可以限定五行。不过，更好的做法如下：

```
SELECT common_name, scientific_name, family_id
FROM birds
WHERE family_id IN(103, 160, 162, 164)
AND common_name != ''
ORDER BY common_name
LIMIT 3, 2;

+------------------------+------------------------+-----------+
| common_name            | scientific_name        | family_id |
+------------------------+------------------------+-----------+
| American Avocet        | Recurvirostra americana |       162 |
| American Golden-Plover | Pluvialis dominica     |       103 |
+------------------------+------------------------+-----------+
```

这个 LIMIT 子句带了两个值：一个是开始位置，一个是行数。结果是第 3 行和第 4 行。顺便说一下，之前用的 LIMIT 3 其实可以写成 LIMIT 0, 3：0 能保证不跳过任何行。

7.5　表连接

从本章开始到现在，我们每次都只操作一个表。现在就来看看一些从多个表中获取数据的方法。这时，我们得指示 MySQL，需要从哪些表抓数据，以及如何连接这些表。

例如，试试查询一些鸟种及其各自对应的科。为了简单起见，我们就查同一目而不同科的鸟。在前面查询岸鸟的例子中，我们看到那些鸟的 order_id 都是 102。那我们就直接用这

个。输入以下 SELECT 语句：

```
SELECT common_name AS 'Bird',
bird_families.scientific_name AS 'Family'
FROM birds, bird_families
WHERE birds.family_id = bird_families.family_id
AND order_id = 102
AND common_name != ''
ORDER BY common_name LIMIT 10;

+-----------------------+------------------+
| Bird                  | Family           |
+-----------------------+------------------+
| African Jacana        | Jacanidae        |
| African Oystercatcher | Haematopodidae   |
| African Skimmer       | Laridae          |
| African Snipe         | Scolopacidae     |
| Aleutian Tern         | Laridae          |
| Amami Woodcock        | Scolopacidae     |
| American Avocet       | Recurvirostridae |
| American Golden-Plover | Charadriidae    |
| American Oystercatcher | Haematopodidae  |
| American Woodcock     | Scolopacidae     |
+-----------------------+------------------+
```

此 SELECT 语句返回 birds 表和 bird_families 表的各一列。这条语句有点长，别急。它跟本章之前的那些语句是相似的，大的改动只有一处，其他都只是轻微变化。我们首先关注一下那个大改动：它如何帮助我们从两个表中获取数据。

在 FROM 子句中，我们用逗号分隔，列出了两个表。而在 WHERE 子句中，我们指定了所需的行，是在另一个表中能找到相同 family_id 的那些。如果不这样指定，结果中就会出现重复的行。而因为两个表都有 family_id 列，为避免名字混淆，我们需要在列名前加上表名，并且表名和列名之间用一个点来分隔（即 birds.family_id）。对于 SELECT 后的学名列，也是这种做法（bird_families.scientific_name）。如果不这么做，MySQL 会搞不清我们想要的是 birds 的 scientific_name 还是 bird_families，并导致以下报错信息：

```
ERROR 1052 (23000): Column 'scientific_name' in field list is ambiguous
```

你可能还注意到 SELECT 中多了个新东西：AS 关键字。它用于为结果集的标题声明一个替代名称，或者说别名。如果科名那列没用 AS，那么标题就会显示为 bird_families.scientific_name。这样就不够好看了。这可以说是另一种风格，接下来，我们还会看到它的实用之处。除了列名，AS 还可为表起别名，如下所示：

```
SELECT common_name AS 'Bird',
families.scientific_name AS 'Family'
FROM birds, bird_families AS families
WHERE birds.family_id = families.family_id
AND order_id = 102
AND common_name != ''
ORDER BY common_name LIMIT 10;
```

此例中，我们为 bird_families 起了一个更短的名字，那就是 families。注意，表的别名

不能加引号。

一旦给表声明了别名，整个语句里用到这个表的地方，都得使用该别名。所以，SELECT 中的 bird_families.scientific_name 需改为 families.scientific_name，WHERE 中的 bird_families.family_id 需改为 families.family_id。如果不改，就会有以下报错信息：

```
ERROR 1054 (42S22):
Unknown column 'bird_families.family_id' in 'where clause'
```

现在，给该语句加上第三个表，以获取每种鸟的目的名称。做法如下：

```
SELECT common_name AS 'Bird',
families.scientific_name AS 'Family',
orders.scientific_name AS 'Order'
FROM birds, bird_families AS families, bird_orders AS orders
WHERE birds.family_id = families.family_id
AND families.order_id = orders.order_id
AND families.order_id = 102
AND common_name != ''
ORDER BY common_name LIMIT 10, 5;

+-----------------+-----------------+-----------------+
| Bird            | Family          | Order           |
+-----------------+-----------------+-----------------+
| Ancient Murrelet | Alcidae        | Charadriiformes |
| Andean Avocet   | Recurvirostridae | Charadriiformes |
| Andean Gull     | Laridae         | Charadriiformes |
| Andean Lapwing  | Charadriidae    | Charadriiformes |
| Andean Snipe    | Scolopacidae    | Charadriiformes |
+-----------------+-----------------+-----------------+
```

来看看它跟之前的语句有什么区别。我们在 FROM 中加入了第三个表，并且给了它一个别名(orders)。为了正确地连接第三个表，我们在 WHERE 中增加了一个运算逻辑：families.order_id = orders.order_id。这使得 SELECT 能返回与 families 相应的 orders 的行的 scientific_name 列。同时，我们还在 SELECT 中加上了它的 scientific_name 列，以显示目的学名。因为我们所查的科都属于同一目，所以这个查询结果看起来没什么意义。不过我们未来也可能需要查不同的目，而这个语句到时还是适用的。最后，我们在 LIMIT 中指定了起始行数，以查得接下来的五行。

> 如果列的别名就是一个单词，那么没有必要加引号。否则，要加。另外，保留字（如 Order）也要加引号。

7.6 表达式与LIKE

为了查出更多的目，我们来改改刚才的 SELECT 语句。先看看 WHERE 中关于 common_name 的条件：

```
AND common_name != ''
```

我们改一下这个匹配条件，换成 LIKE 运算符（我们在第 6 章见过），使之查出各种相似的俗名。虽然所属的科不同，但有些鸟的体型大小是相近的。最小的那些鸟，通常俗名中都带有 Least 这样的字眼，于是我们就查查俗名中含有 Least 的鸟：

```
SELECT common_name AS 'Bird',
families.scientific_name AS 'Family',
orders.scientific_name AS 'Order'
FROM birds, bird_families AS families, bird_orders AS orders
WHERE birds.family_id = families.family_id
AND families.order_id = orders.order_id
AND common_name LIKE 'Least%'
ORDER BY orders.scientific_name, families.scientific_name, common_name
LIMIT 10;

+------------------+---------------+------------------+
| Bird             | Family        | Order            |
+------------------+---------------+------------------+
| Least Nighthawk  | Caprimulgidae | Caprimulgiformes |
| Least Pauraque   | Caprimulgidae | Caprimulgiformes |
| Least Auklet     | Alcidae       | Charadriiformes  |
| Least Tern       | Laridae       | Charadriiformes  |
| Least Sandpiper  | Scolopacidae  | Charadriiformes  |
| Least Seedsnipe  | Thinocoridae  | Charadriiformes  |
| Least Flycatcher | Tyrannidae    | Passeriformes    |
| Least Bittern    | Ardeidae      | Pelecaniformes   |
| Least Honeyguide | Indicatoridae | Piciformes       |
| Least Grebe      | Podicipedidae | Podicipediformes |
+------------------+---------------+------------------+
```

上例中，MySQL 根据 LIKE 的条件，筛选出了开头为 Least，结尾为任意内容（% 通配符）的 common_name。另外，因为我们去掉了 families.order_id = 102 子句，所以查出的鸟种不限于一个目。你可以从结果中看到有不同的目名。

同时，我们还修改了 ORDER BY 子句，使 MySQL 在临时表中将结果先按目的学名排序，再按科的学名排序，最后按种的俗名排序。看看最终出来的结果，你就会知道是怎么回事：首先是按目排序。然后，看看目为 Charadriiformes 的那些行，你会发现科按字母序排列了。而对于科名同为 Caprimulgidae 的那两行，是按种的俗名来排序的。

在 ORDER BY 中不能使用列的别名，但表的别名则可以。事实上，如果在 FROM 中指定了表的别名，那么在 ORDER BY 中就必须用这个别名。

刚才我们是使用 LIKE 来做模式匹配的。除此之外，还可以使用 REGEXP，它支持更多样的模式。现在就用它来改写上面的语句。小鸟的俗名通常以 Least 开头，上例就是以此查询小鸟的。而一科中最大的那些鸟，通常叫 Great。为了同时查出小鸟和大鸟，我们使用以下 SQL 语句：

```
SELECT common_name AS 'Birds Great and Small'
FROM birds
WHERE common_name REGEXP 'Great|Least'
ORDER BY family_id LIMIT 10;

+----------------------------+
| Birds Great and Small      |
+----------------------------+
| Great Northern Loon        |
| Greater Scaup              |
| Greater White-fronted Goose |
| Greater Sand-Plover        |
| Great Crested Tern         |
| Least Tern                 |
| Great Black-backed Gull    |
| Least Nighthawk            |
| Least Pauraque             |
| Great Slaty Woodpecker     |
+----------------------------+
```

这里，REGEXP 所带的表达式是用引号包围起来的两个值：Great 和 Least。MySQL 的默认匹配方法，是将 REGEXP 所带的文本与列中值的开头开始匹配。若想强制在头部进行匹配，可在表达式的前端加上克拉符号（即 ^），不过这里不需要。两个表达式之间的竖线（即 |），表示可接受两种值，即“或”的意思。

你会发现，结果除了包含 Great 开头的俗名，还包含了一些以 Greater 开头的俗名。如果想排除 Greater，可以使用 NOT REGEXP 运算符，如下所示：

```
SELECT common_name AS 'Birds Great and Small'
FROM birds
WHERE common_name REGEXP 'Great|Least'
AND common_name NOT REGEXP 'Greater'
ORDER BY family_id LIMIT 10;

+-------------------------+
| Birds Great and Small   |
+-------------------------+
| Great Northern Loon     |
| Least Tern              |
| Great Black-backed Gull |
| Great Crested Tern      |
| Least Nighthawk         |
| Least Pauraque          |
| Great Slaty Woodpecker  |
| Great Spotted Woodpecker |
| Great Black-Hawk        |
| Least Flycatcher        |
+-------------------------+
```

NOT REGEXP 消除了所有的 Greater 鸟。注意，我们要用 AND 来把它跟其他条件并起来，而不是将它写在原本那个 REGEXP 中。

顺便说一下，这里是按 family_id 来排序的，这使得同科的鸟被排在一起。不过鸟的名字

则没有顺序，看起来怪怪的。当然，我们可以在 ORDER BY 中加上 common_name，使得同一科的鸟按字母序排列。

REGEXP 和 NOT REGEXP 不区分大小写。如果想区分大小写，则需要加上 BINARY 选项。现在我们用另一组鸟来做例子，讲讲这个问题。这次查询的是 Hawks，注意它是以大写字母开头的。具体来说，结果必须要是"鹰"，即使名字里有 hawk 的字眼，但如果不是鹰，那也不能算数。例如，Nighthawks 和 Hawk-Owls 都不合格。因为在 birds 表中，俗名里的每个单词都以大写字母开头（像标题一样），所以，使用 Hawk 这样的模式（首字母大写，其余小写），并加上 BINARY 选项，就可以剔掉 Nighthawks 之类的名称了。另外，我们还要用 NOT REGEXP 来排除 Hawk-Owls。现在试试：

```
SELECT common_name AS 'Hawks'
FROM birds
WHERE common_name REGEXP BINARY 'Hawk'
AND common_name NOT REGEXP 'Hawk-Owl'
ORDER BY family_id LIMIT 10;

+-------------------+
| Hawks             |
+-------------------+
| Red-tailed Hawk   |
| Bicolored Hawk    |
| Common Black-Hawk |
| Cuban Black-Hawk  |
| Rufous Crab Hawk  |
| Great Black-Hawk  |
| Black-faced Hawk  |
| White-browed Hawk |
| Ridgway's Hawk    |
| Broad-winged Hawk |
+-------------------+
```

前面说过，REGEXP 和 NOT REGEXP 不区分大小写，除非加上 BINARY 选项，如上例，这就指定了它按二进制来对比（因为 H 的二进制表示与 h 的是不同的）。但对于 common_name 列，其实不必要指定 BINARY 选项，因为它已经使用了二进制校对方式。这是我们在第 4 章开头建立 rookery 库时，无意识设定的。现在，用以下语句来看看 rookery 库是怎样建的：

```
SHOW CREATE DATABASE rookery \G

*************************** 1. row ***************************
       Database: rookery
Create Database: CREATE DATABASE `rookery` /*!40100 DEFAULT
                 CHARACTER SET latin1 COLLATE latin1_bin */
```

其中 COLLATE 子句设置了 latin1_bin，意思是 Latin1 二进制。因此，除非在建表时另外声明，否则建在 rookery 库中的表都会遵循 latin1_bin 这一设定。再看看 birds 表的 common_name 是如何设定的：

```
SHOW FULL COLUMNS
FROM birds LIKE 'common_name' \G
```

```
*************************** 1. row ***************************
     Field: common_name
      Type: varchar(255)
 Collation: latin1_bin
      Null: YES
       Key:
   Default: NULL
     Extra:
Privileges: select,insert,update,references
   Comment:
```

以上语句只会显示 common_name 的相关信息。注意，它的 Collation 是 latin1_bin。因此，对它使用 REGEXP 进行正则表达式匹配时，就是区分大小写的，无需再指定 BINARY。

再检查一下 birds 表，我们发现有些鸟的俗名是没有横杠的，如 Hawk Owl。这样就没法符合我们刚才给的 Hawk-Owl 条件了。另外，还有些名字中的 Hawk 并不是首字母大写，所以不能依赖大小写判断。上面的代码已经漏掉了某些鸟，所以我们需要修改一下表达式：

```
SELECT common_name AS 'Hawks'
FROM birds
WHERE common_name REGEXP '[[:space:]]Hawk|[[.hyphen.]]Hawk'
AND common_name NOT REGEXP 'Hawk-Owl|Hawk Owl'
ORDER BY family_id;
```

这个很长的 REGEXP 表达式使用了*字符类*和*字符名*。它们的书写格式是，将字符类置于两个方括号中，用冒号包围（如 [[:alpha:]] 表示字母）；字符名则使用方括号和点号包围（如 [[.hyphen.]] 表示横杠）。你应该能猜到，我们要找的是 common_name 含有“ Hawk”或“-Hawk”的行——即在 Hawk 前，要有一个空格或横杠。对于 Hawk 前是字母的那些名称（如 Nighthawk），是不合格的。而第二个表达式则用于排除 Hawk-Owl 和 Hawk Owl。

MySQL 的正则表达式比其他语言如 Perl 和 PHP 更冗长。不过，它们对于一些基本要求还是可行的。对于复杂的正则表达式，你可能会想使用 Perl DBI 之类的 API，来在 MySQL 外面进行数据处理。但这样性能会不如先在数据库内进行过滤，所以最好还是在 MySQL 里使用 REGEXP。

7.7 对结果集进行计数和分组

在之前的很多示例中，因为结果可能含有几千行，所以我们都限定了返回的行数。假设我们想知道具体有多少行，可以在语句中加一个函数，即 COUNT()。来看看它的使用方法：

```
SELECT COUNT(*) FROM birds;

+----------+
| COUNT(*) |
+----------+
|    28891 |
+----------+
```

我们在括号中填入星号，以表示需要统计所有行。如果不填星号，而填某一列名，则表示只统计有值的行，即让 MySQL 忽略列中是 NULL 值的行。但注意，空或空值（即 ''）并

不会被忽略。

知道整个 birds 表有多少行是不错的。但如果只想统计某部分呢？试试用 COUNT() 来计算某一科（如 Pelecanidae，即 Pelican）的行数。输入以下 SQL 语句：

```
SELECT families.scientific_name AS 'Family',
COUNT(*) AS 'Number of Birds'
FROM birds, bird_families AS families
WHERE birds.family_id = families.family_id
AND families.scientific_name = 'Pelecanidae'

+-------------+-----------------+
| Family      | Number of Birds |
+-------------+-----------------+
| Pelecanidae |              10 |
+-------------+-----------------+
```

如你所见，Pelecanidae 科有 10 种鸟。得出这样的结果是因为我们在 WHERE 子句中限制了 Pelecanidae 的查询条件。如果想得知 Pelican 所属的目（Pelecaniformes）的鸟类总数，可以在上例中连接 bird_orders 表。在 mysql 客户端中输入以下命令：

```
SELECT orders.scientific_name AS 'Order',
families.scientific_name AS 'Family',
COUNT(*) AS 'Number of Birds'
FROM birds, bird_families AS families, bird_orders AS orders
WHERE birds.family_id = families.family_id
AND families.order_id = orders.order_id
AND orders.scientific_name = 'Pelecaniformes';

+----------------+-------------+-----------------+
| Order          | Family      | Number of Birds |
+----------------+-------------+-----------------+
| Pelecaniformes | Pelecanidae |             224 |
+----------------+-------------+-----------------+
```

结果显示共有 224 种鸟属于 Pelecaniformes 这个目。但科名那一列只显示了第一个找到的科。如果想要得出每个科的名字和鸟种数量，就需要 MySQL 对结果集进行分组。你得告诉 MySQL 要根据哪一列来分组。这时，就要靠 GROUP BY 子句了。它所带的值就是分组的依据。现在看看它的用法，输入以下语句：

```
SELECT orders.scientific_name AS 'Order',
families.scientific_name AS 'Family',
COUNT(*) AS 'Number of Birds'
FROM birds, bird_families AS families, bird_orders AS orders
WHERE birds.family_id = families.family_id
AND families.order_id = orders.order_id
AND orders.scientific_name = 'Pelecaniformes'
GROUP BY Family;

+----------------+-------------------+-----------------+
| Order          | Family            | Number of Birds |
+----------------+-------------------+-----------------+
| Pelecaniformes | Ardeidae          |             157 |
| Pelecaniformes | Balaenicipitidae  |               1 |
```

```
| Pelecaniformes | Pelecanidae       |               10 |
| Pelecaniformes | Scopidae          |                3 |
| Pelecaniformes | Threskiornithidae |               53 |
+----------------+-------------------+-----------------+
```

此例中，GROUP BY 子句带有 Family 这个别名，它是指 bird_families 表的 scientific_name 列。MySQL 根据这一条语句，返回了含有五行的结果集。

GROUP BY 很有用，且很常用，所以请掌握好它。第 12 章将更详细地介绍该子句及相关函数。

7.8 小结

SELECT 语句的玩法实在太多了，而本书只是入门教程，所以我只好省略掉一些，以免搞得太过高深。SELECT 语句有多个缓存结果集的选项，也有将结果集导出到文本文件中的子句。如果你需要用到它们，可以从他处获取相关知识。

暂时来说，你得先确保自己已掌握 SELECT 语句及其主要组成部分：选取列并为其设置别名；在 FROM 子句中指定多个表；构建 WHERE 子句，包括使用正则表达式；使用 ORDER BY 和 GROUP BY 子句；用 LIMIT 子句限制返回部分结果集。想要完全熟悉以上内容，需要一段时间的练习。在进入第 8 章之前，先完成以下习题吧。

7.9 习题

以下习题能使你加深对 SELECT 语句的理解。输入 SQL 语句，尤其是像 SELECT 这种常用的语句，能帮助你学习、记忆和熟悉它们。

(1) 组织一条 SELECT 语句，从 birds 表查出各种鸟的俗名。要求使用 LIKE 来只选出 Pigeons。还有，给 common_name 起个别名 Bird，并使结果集按该列排序。不限制结果集返回行数，让 MySQL 查出全部符合条件的记录。执行此语句，看看结果如何。

    接着，给该语句加上 LIMIT 子句，使其只显示前十行记录。将执行结果和之前的 SELECT 比较，看看是否是前十行。然后改成显示下十行，执行后再和第一次的结果比较，看得到的是否是第 11 行到第 20 行。如果有错，则检查并修改，直至正确。

(2) 本题会以一个简单的 SELECT 语句开始，然后将其变得复杂。首先，组织一条 SELECT 语句，从 bird_orders 表查出 scientific_name 列和 brief_description 列。其中，scientific_name 的别名为 Order——记得加上引号，因为它是保留字。而 brief_description 的别名则为 Types of Birds in Order。无需限制结果集。写好后，执行该语句。若有报错，试着查找原因并修复，直至执行成功。

    再写一个查 birds 表的 SELECT 语句。要求查出 common_name 列和 scientific_name 列。各起别名为 Common Name of Bird 和 Scientific Name of Bird。如果某一行的 common_name 为空，则去掉此行。结果按 common_name 排序。最终限制显示 25 行数据。执行语句并确保无误。

    将以上两条语句合并，使你能从那两个表中取到那四列的数据，并且别名不变（相关技巧在 7.5 节中讲过）。你需要在 FROM 子句中声明多个表，并且在 WHERE 子句中提供表连

接的条件。这道题有点绕。你得花点时间，千万不要气馁。如果有必要，也可以重温 7.5 节。

将最终结果限制为 25 行。如果编写正确，那么出来的鸟种应该会与之前单查 `birds` 表所得的结果一致。记得剔除 `common_name` 列为空的行。

(3) 使用在 `WHERE` 子句中带有 `REGEXP` 的 `SELECT` 语句，从 `birds` 表中获取 `common_name` 含有 Pigeon 或 Dove 字眼的行（相关内容在 7.6 节中讲过）。然后，给 `common_name` 起个别名 `Type of Columbidae`——因为 Dove 和 Pigeon 都属于 Columbidae 科。

第 8 章

更新和删除数据

数据库中的数据是会经常改变的。有时可能是添加一点信息，有时可能是删除某条记录。对于修改或添加某部分数据，主要是用 UPDATE 语句。而对于删除整条记录，基本上就是用 DELETE 语句。本章将详细介绍这两个 SQL 语句。

8.1 更新数据

UPDATE 语句用于修改现存记录的某些列里的值。其基本语法就是 UPDATE 关键字，后接一个表名，再接一个 SET 子句。一般来说，我们还会加上一个 WHERE 子句，以使它只更新某些行。以下是 UPDATE 的简单例子：

```
UPDATE table
SET column = value, ... ;
```

此语法与明确插入的 INSERT 语句很像，它们都有 SET 子句。但与 INSERT 不同的是，UPDATE 没有“非明确”语法。还有一个区别，就是 UPDATE 没有 INTO 子句，表名是紧跟着 UPDATE 关键字的。

现在来看看 UPDATE 的一个例子。第 5 章创建了一个 birdwatchers 数据库，并在其中建了一个 humans 表，以保存 rookery 网站的观鸟爱好者的信息。后来我们在该表中录入了一些人的信息。而在该章结尾的一个习题中，我们给该表加了一列 country_id，以保存会员所在国家的国家代码。假设我们所录入的那少量用户，他们都住在美国。虽然我们可以设置 country_id 默认值为美国（即 us），但其实我们希望大部分会员来自欧洲。所以，我们就先将该表的 country_id 全部设置成 us。执行以下命令：

```
UPDATE birdwatchers.humans
SET country_id = 'us';
```

此语句会将表中所有行的 country_id 都设为 us。此前它们都是 NULL。而即使之前它们有其他值（其他国家代码），这些值也会被改成 us。这将造成非常大的影响。一旦执行，基本上是没有方法去撤销的——除非你的表是 InnoDB 的，并且你是在事务中执行此语句。所以，使用 UPDATE 要小心。你应加上 WHERE 来指定要更新的行，并在实际更新之前先做测试，测试方法我们很快就会看到。

注意，上例还指定了数据库名称，那是因为在前面章节中，我们在 mysql 中将 rookery 设为了默认数据库。而本章所有示例都是用 birdwatchers 的，所以现在用 USE 来将默认数据库改为它吧：

```
USE birdwatchers;
```

为了执行本章接下来的示例，你最好先到 MySQL 资源站（http://mysqlresources.com/files）下载 rookery 和 birdwatchers 的 dump 文件。它们所含的数据量更大，更适合练习。

8.1.1 更新指定行

大多数情况下，我们用 UPDATE 时都会带上 WHERE 子句，以指定哪些行会按 SET 子句更新。UPDATE 的 WHERE 子句的条件，与 SELECT 的一样。事实上，正因为一样，所以你可以先用 SELECT 来测试 WHERE 的条件，再应用到 UPDATE 中。我们很快就会在本章中看到具体做法。现在先来看看如何只更新一行。

humans 表中有一行是一位叫 Rusty Osborne 的女士。她最近结婚了，需要将姓改为丈夫的姓——Johnson。我们可以用 UPDATE 来处理。首先，获取她那条记录。这需要根据她的姓和名来查询，因为姓 Osborne 的人可能有很多，而全名为 Rusty Osborne 的应该就只有她。在 mysql 中输入以下语句：

```
SELECT human_id, name_first, name_last
FROM humans
WHERE name_first = 'Rusty'
AND name_last = 'Osborne';

+----------+------------+-----------+
| human_id | name_first | name_last |
+----------+------------+-----------+
|        3 | Rusty      | Osborne   |
+----------+------------+-----------+
```

从结果可以看出，我们确实只有一个 Rusty Osborne，她的 human_id 为 3。于是我们就可以在 UPDATE 中用这个 ID，以确保只更新这一行。输入以下语句：

```
UPDATE humans
SET name_last = 'Johnson'
WHERE human_id = 3;

SELECT human_id, name_first, name_last
FROM humans
WHERE human_id = 3;
```

```
+----------+-----------+-----------+
| human_id | name_first | name_last |
+----------+-----------+-----------+
|        3 | Rusty      | Johnson   |
+----------+-----------+-----------+
```

成功了。使用 UPDATE 是很简单的，尤其是当你已经知道了所要更新的行的键值时。现在，假设有两个已婚的女会员希望我们将其头衔从 Mrs. 改为 Ms.（此信息保存在枚举列 formal_title 中）。经过 SELECT 查找，我们得知她们的 human_id 分别为 24 和 32。于是，可以使用以下 UPDATE 语句：

```
UPDATE humans
SET formal_title = 'Ms.'
WHERE human_id IN(24, 32);
```

当想要更新的行数多于一行时，就要用到稍微复杂一点的写法了。不过，只要知道键值，仍然很简单。就像本例，我们用了 IN 运算符来列出需要匹配的多个 human_id 号。

我们决定与时俱进，除了修改这两个人的头衔，还打算将所有已婚女会员的头衔都改成 Ms.。我们还是用 UPDATE，但这次要修改一下 WHERE 子句。formal_title 为 Mrs. 的女会员可能有很多，如果全部手动输入她们的 human_id，不太现实。我们需要更简单的做法。首先，看看 formal_title 列是怎样的：

```
SHOW FULL COLUMNS
FROM humans
LIKE 'formal_title' \G


*************************** 1. row ***************************
     Field: formal_title
      Type: enum('Mr.','Miss','Mrs.','Ms.')
 Collation: latin1_bin
      Null: YES
       Key:
   Default: NULL
     Extra:
Privileges: select,insert,update,references
   Comment:
```

该列的枚举值看起来有点性别歧视。男孩和男人，不管年龄和婚姻状况，都只有一种称呼，但女性却有三种。而且，我们缺少不指明性别的称呼，例如 Dr.，不过，我们还是先不考虑这个。事实上，我们也可以完全去掉这列以消除性别歧视，但别急着这么做。现在先改表结构，使该列只含两个选项：Mr. 和 Ms.。而在改表结构之前，我们得先修改已填充的值。为了达到目的，要用 UPDATE 语句：

```
UPDATE humans
SET formal_title = 'Ms.'
WHERE formal_title IN('Miss','Mrs.');
```

把现存记录的 formal_title 都改成 Mr. 或 Ms. 之后，就可以更改该列的设置，去掉其他选项。这将用到第 4 章讲过的 ALTER TABLE 语句。输入以下命令执行更改：

```
ALTER TABLE humans
CHANGE COLUMN formal_title formal_title ENUM('Mr.','Ms.');

Query OK, 62 rows affected (0.13 sec)
Records: 62  Duplicates: 0  Warnings: 0
```

从结果中的返回信息可以看到，更改顺利。不过，如果你在改表结构之前，有一行数据忘了修改（例如有个人的 Miss 没改成 Ms.），那么返回信息中的 Warnings 就会是 1。出现这种情况时，可以使用 SHOW WARNINGS 来查看该警告：

```
SHOW WARNINGS \G

*************************** 1. row ***************************
  Level: Warning
   Code: 1265
Message: Data truncated for column 'formal_title' at row 44
```

此警告的意思是，MySQL 清除了第 44 行的 formal_title 列的值。于是我们还得再用 UPDATE 来修复该行的头衔，并且新头衔也要符合新的枚举值。所以，多数情况下，改表结构前，最好先更新数据。

不过也有反过来的时候：在修改大批数据时，要在更新前先改表结构。例如，假设我们想将 formal_title 的枚举值改成 Mr 和 Ms（没有句点）。这种情况下，可以先给它的枚举列表加上这两个新选项，然后再将数据改成新值。而 UPDATE 语句的 WHERE 子句也要修改。因为新值与旧值存在这样的对应关系：新值是旧值的前两个字符，所以可以在 UPDATE 中使用截取函数来做更新。大概如下：

```
ALTER TABLE humans
CHANGE COLUMN formal_title formal_title ENUM('Mr.','Ms.','Mr','Ms');

UPDATE humans
SET formal_title = SUBSTRING(formal_title, 1, 2);

ALTER TABLE humans
CHANGE COLUMN formal_title formal_title ENUM('Mr','Ms');
```

第一个 ALTER TABLE 语句只是给 formal_title 加了两个不带句点的新选项，并没有去除原选项，这是因为现存的数据在使用这两个旧值。最后的 ALTER TABLE 才是去除原选项。这两个语句没什么特别。而第二个语句的 UPDATE 就有趣多了。

在 SET 子句中，我们将 formal_title 的值设为了当前值的子串，其中用到 SUBSTRING() 函数来抽取文本内容。在其括号中，我们首先给出了子串的来源（formal_title），然后是子串的起始位置：1，表示原字符串的第一个字符；接着，指定需要抽取的字符数量：2。所以，SUBSTRING() 遇到 Mr. 时，会抽出 Mr，而遇到 Ms. 时，会抽出 Ms。

注意，函数不会改变原数据。SUBSTRING() 只是返回子串。若想落实修改，还需要 SET formal_title = 子句。这样才能将 SUBSTRING() 得出的值应用到 formal_title 中。还有，如果需要的话，甚至可以对一列进行 SUBSTRING() 运算，然后将其返回值用于更新另一列。

本章会用到不少有益于 UPDATE 的字符串函数。想了解更多的字符串函数，详见第 10 章。

8.1.2　按行数更新

本章开头曾经提到，UPDATE 语句很强大，它能快速更改大量数据。所以，几乎所有时候，你都应该给它带上 WHERE 子句，以便用条件限制它所能更新的行。有时你可能还想用行数来限制更新，可以用 LIMIT。它在 UPDATE 中的作用与在 SELECT 中一样，不过目的不一样。而为什么你会在 UPDATE 中用 LIMIT，以及如何使用它，下面我们就用例子来说明。

假设我们决定每月挑选两个会员来给他们一些奖励，以吸引人们加入我们的网站。奖励可能是一本野外鸟类摄影图册，可能是一支上面有 Rookery 字样的钢笔，也可能是一个有鸟只图案的水瓶。再假设每个会员都有且只有一次机会获奖。为了记录中奖者，我们得创建一个表来保存每个中奖者的姓名和中奖时间，以及具体奖品和发奖时间。用以下的 CREATE TABLE 语句来建这个表：

```
CREATE TABLE prize_winners
(winner_id INT AUTO_INCREMENT PRIMARY KEY,
 human_id INT,
 winner_date DATE,
 prize_chosen VARCHAR(255),
 prize_sent DATE);
```

此语句将创建一个名为 prize_winners 的表，其中会有五列：第一列（winner_id）是每行的标识号；第二列（human_id）用于关联 humans 表；第三列（winner_date）用于保存抽奖时间；接着的 prize_chosen 列保存所选的奖品；最后的 prize_sent 列保存奖品发出的时间。

> 此表的两个 ID 可能令人困惑。winner_id 用于标识表中每条记录，可以用它来查出每次抽奖的奖品和日期。而 human_id 则是用于从 humans 表查询每个中奖者的信息。因为每个人只有一次中奖机会，所以你可能会觉得没必要用两个 ID。但回想之前我们在 birds、bird_families 和 bird_orders 之间进行表连接的方式，就会发现，给每个表都指定一个专属的 ID 列，是一种健壮的设计。

prize_chosen 列也可以设为枚举类型，但未来奖品可能有更多选择，最终我们可能还得再创建一个表来保存奖品信息，然后将 prize_chosen 改成外键，参考奖品表。所以我们暂时就将它设为足够长的变长字符串类型吧。

因为每个会员都有一次中奖机会，所以我们可以先将所有会员都导入 prize_winners。不这样做的话，也可以延迟到抽奖时才把中奖者录入。这对于数据维护来说应该更好，但因为 INSERT...SELECT 很便捷，所以我们就直接 INSERT...SELECT 吧（6.3.2 节介绍过这种语法）：

```
INSERT INTO prize_winners
(human_id)
SELECT human_id
FROM humans;
```

此语句将 humans 表的每个会员都插入 prize_winners 了。我们只填写 human_id 列，因为现

在还没抽奖。另外，因为 winner_id 是 AUTO_INCREMENT 的，所以它被自动填写了一系列递增且每一行都不同的值。winner_id 的值不应与 human_id 一样，因为 human_id 另有用途。除了这两列，其他列暂时都是 NULL。有人获奖时，我们才会去更新这些列。

既然已经建好了中奖表，那么我们就开始抽奖吧。

8.1.3 排序后再按行数更新

在前一节中，我们说要通过为会员发放奖品来吸引新用户加入网站，并维持与老会员的关系。我们会让 MySQL 每个月随机抽选会员，使新老会员都有同等的获奖机会。具体做法就是用 UPDATE 语句，并带上 ORDER BY 子句和 RAND() 函数。此函数在这里的作用是，给符合 WHERE 条件的每一行分配一个随机的浮点数。然后，因为我们把它加入 ORDER BY 了，所以结果集就按照这个随机数来排序了。如果再在后面加上 LIMIT 2，那么就可以每次都只返回两个获奖者：

```
UPDATE prize_winners
SET winner_date = CURDATE()
WHERE winner_date IS NULL
ORDER BY RAND()
LIMIT 2;
```

不过 RAND() 是有缺陷的，它其实并不是很随机，有时会返回相同数值。所以你得考虑清楚它是否真的适用于你的需求。

我们从底往上分析这个语句吧。ORDER BY 在这里有点讽刺，因为它按 RAND() 所造出的乱序来排序。而 LIMIT 则将结果集限制为两行。因此，人人都有平等机会成为两个中奖者之一。

不过，这还不足以保证查出来的两个人是未中过奖的。我们有可能因为乱序而在不同的月份抽到相同的人。所以，要 UPDATE 的是用 WHERE 来选出 winner_date 为 NULL 的行，即那些没中过奖的人。最后，在 SET 中，将 winner_date 设为当前日期。这里所用到的 CURDATE()，会在第 11 章介绍。

此外，这个语句还有些不太明显的问题。首先，在 ORDER BY 中使用 RAND() 可能有性能问题。如果你的表不大，那可能察觉不到。但如果表很大，服务器又很繁忙，那就可能要运行很久了。所以，在 ORDER BY 中用 RAND() 时，你得考虑表的大小和运行环境。其次，在“MySQL 复制”的环境下，同时使用 ORDER BY 和 LIMIT，也可能导致问题，除非你的复制是基于行的。“MySQL 复制”是一套主从服务器配置，它允许你把主服务器的数据复制到从服务器上。它是一个高级话题，但因为在 UPDATE 中同时使用 ORDER BY 和 LIMIT 可能导致一些问题，所以我还是想说说。它可能导致的问题就是会出现以下警告：

```
SHOW WARNINGS \G

*************************** 1. row ***************************
  Level: Warning
   Code: 1592
Message: Statement is not safe to log in statement format.
```

如果你没有在用“MySQL 复制”，那么可以忽略这个警告。而如果在用，那么就有可能导

致从库的数据与主库的（或其他从库的）不一致——尤其是使用了 RAND()（因为语句应用到从库时所取的随机数不同了）。不过，因为你还处在初学阶段，所以可以暂时忽略这个东西，并且安全地使用子句和这个函数。我只是希望你了解这些潜在的问题，并让你知道 MySQL 的相关知识是多么博大精深。

8.1.4 同时更新多个表

本章直到现在，我们试过用原表中的数据来更新表，但其实取其他表的数据来做更新也是可以的。另外，我们使用的 UPDATE 都是一次只更新一个表，但其实一次更新多个表也是可以的。接下来我们就看看为何以及如何做到这两点。

假设发了一段时间的奖品后，我们决定加大力度，吸引英国人加入并留在我们的网站。于是，我们打算每次发四个奖品，其中两个发给来自英国的会员，另外两个发给英国以外的会员。我们会全站告知这一决定，甚至允许拿过奖的英国用户再拿一次。为了做到这一点，我们得先按 humans 的 country_id 来重置 prize_winners 的数据。下面就是我们的做法：

```
UPDATE prize_winners, humans
SET winner_date = NULL,
    prize_chosen = NULL,
    prize_sent = NULL
WHERE country_id = 'uk'
AND prize_winners.human_id = humans.human_id;
```

此语句先从一个表中查找符合条件的行，然后将这些行与另一个表的行进行连接，最后更新另一个表的那些行。首先，列出两个表，并用逗号分隔。接着，使用 SET 子句，将获奖相关的列设为 NULL。最后在 WHERE 子句中，给出筛选 humans 表的 country_id 条件：含有 uk 这个值，并且两个表按 human_id 连接。

重置完英国用户的获奖记录后，就可以开始这个月的抽奖了。找回之前那个随机挑选用户的 UPDATE 语句，但这次将它改成含有 humans 和 prize_winners：

```
UPDATE prize_winners, humans
SET winner_date = CURDATE()
WHERE winner_date IS NULL
AND country_id = 'uk'
AND prize_winners.human_id = humans.human_id
ORDER BY RAND()
LIMIT 2;

ERROR 1221 (HY000): Incorrect usage of UPDATE and ORDER BY
```

你可能认为这么写是可以的，但其实不行。语句执行失败，并返回了一个报错信息。那是因为，在 MySQL 使用 UPDATE 的多表语法时，不能带有 ORDER BY 或 LIMIT——但在 UPDATE 单表时就可以。这令人沮丧，不过我们还是有办法解决它。因为 UPDATE 单表可以用 ORDER BY、RAND() 和 LIMIT，所以我们可以先在子查询（即包含在查询语句中的另一个查询）中，从 humans 表随机选出得奖者，然后按他们来更新 prize_winners 表。下面就来看看实现方法：

```
UPDATE prize_winners
SET winner_date = CURDATE()
WHERE winner_date IS NULL
AND human_id IN
  (SELECT human_id
   FROM humans
   WHERE country_id = 'uk'
   ORDER BY RAND())
LIMIT 2;
```

这好像很复杂，但分解开来看，就会发现其实并不难。首先看看那个内嵌的查询，即被括号包围的那个 SELECT 语句。它的作用是查询 humans 表里所有 country_id 为 uk 的会员，并以乱序返回结果集。注意，我们查找的是所有英国用户，而且并不区分是否拿过奖。这是因为内嵌的查询语句不能查 UPDATE 语句所作用的表。所以，查询条件需要分开：只更新 winner_date 为 NULL 的语句，要放在 UPDATE 的 WHERE 子句中。整个语句是针对英国用户的。如果是查找英国以外的用户，很简单，只需将子查询中的运算符改成 != 即可。

而在 UPDATE 中，我们用 IN 来限定 prize_winners 的 human_id 必须符合子查询，才会被更新。最后的 LIMIT 指定只更新两行。注意，它属于 UPDATE，而不属于子查询（即 SELECT）。

因为子查询与 UPDATE 是分开的，它被首先执行，所以在里面用 ORDER BY 是没问题的。而 LIMIT 所在的 UPDATE 并没有使用多表更新的语法，所以也能正常运行。

这样写可能看起来很麻烦，但它解决了问题。如果有什么方法你觉得可行，但实际却不行，可以试试用子查询来做。第 9 章将详细介绍子查询。

8.1.5 处理重复

第 6 章已经详细介绍过 INSERT 语句。我们见过它的各种语法和有趣的使用方式，包括 INSERT...SELECT，即 INSERT 和 SELECT 合在一起使用。其实，还有一种组合方式，可用于更新行，那就是 INSERT...ON DUPLICATE KEY UPDATE。

插入多行数据时，你可能会不小心录入重复的数据：即本应各有不同的行，出现了相同值。如果你用的是 INSERT，可以加上 IGNORE 标识，以指示数据库忽略重复的行，不插入它们。而如果用的是 REPLACE，MySQL 就会更新已存在的行，或者说把它们删掉，保留新行。除了以上做法之外，你可能会想保留原本的行，但给它们做上标记。那么，可以用 INSERT...ON DUPLICATE KEY UPDATE。看看下面的例子，你就会明白我的意思。

假设存在另一个观鸟网站，它与我们的网站类似，但名字叫作 Better Birders。不过很少人访问那个网站，于是它的站长想把它关掉，并且他告诉我们，如果我们吸纳那个网站的会员，他就让该站重定向到我们这边。我们同意了。接着，他给了我们一个含有该站会员姓名和电子邮件地址的文本文件。有多种方法导入这个文件，第 15 章将介绍其中一些方法。不过，因为文件中有些人已经在我们的网站上注册过了，所以我们不想重复导入。然而，我们又想把这些会员标记出来，以备日后需要。可以尝试使用 INSERT...ON DUPLICATE KEY UPDATE。首先，用 ALTER TABLE 加一列，用于记录某条会员信息是否来自 Better Birders 网站：

```
ALTER TABLE humans
ADD COLUMN better_birders_site TINYINT DEFAULT 0;
```

此语句增加了名为 better_birders_site 的一列，并设其默认值为 0。对于来自 Better Birders 的记录，其值会被设为 1。而如果是既在 Better Birders，又在我们这边，则记为 2。因为人是有可能同名的，所以我们用电子邮件地址来区分。因为在 humans 表中，我们设置过 email_address 为 UNIQUE，所以 INSERT...ON DUPLICATE KEY UPDATE 就能根据 email_address 来进行更新操作了。下面就动手输入几个会员试试吧：

```
INSERT INTO humans
(formal_title, name_first, name_last, email_address, better_birders_site)
VALUES('Mr','Barry','Pilson', 'barry@gomail.com', 1),
      ('Ms','Lexi','Hollar', 'alexandra@mysqlresources.com', 1),
      ('Mr','Ricky','Adams', 'ricky@gomail.com', 1)
ON DUPLICATE KEY
UPDATE better_birders_site = 2;
```

因为加上了 ON DUPLICATE KEY，所以如果发现有重复的电子邮件地址，better_birders_site 就会被更新为 2。而没有发现重复的那些行，就会记为 1。这正符合我们的要求。

接着，将这些新会员插入 prize_winners 表。我们会跟之前一样，用 INSERT...SELECT 语句，不过只选出 better_birders_site 为 1 的行来插入：

```
INSERT INTO prize_winners
(human_id)
SELECT human_id
FROM humans
WHERE better_birders_site = 1;
```

尽管以上两个语句能按电子邮件地址来区分会员是否是同一个人，但事实上，同一个人也可以用不同的电子邮件地址来注册两个网站。甚至，还可以用不同的电子邮件地址在我们的网站上注册多次。为了检查是否有这种情况并标识出来，我们先做些准备工作：

```
ALTER TABLE humans
ADD COLUMN possible_duplicate TINYINT DEFAULT 0;

CREATE TEMPORARY TABLE possible_duplicates
(name_1 varchar(25), name_2 varchar(25));
```

第一句是给 humans 表增加一列，以标记“可能重复注册”。第二句则是建了一个临时表。临时表只能被同一 MySQL 客户端访问，并且，在退出客户端时，临时表就会被自动删掉。因为在检查重复的过程中，我们不能对原表进行更新，所以需要在临时表里做标记。先用 INSERT...SELECT 查找：

```
INSERT INTO possible_duplicates
SELECT name_first, name_last
FROM
  (SELECT name_first, name_last, COUNT(*) AS nbr_entries
   FROM humans
   GROUP BY name_first, name_last) AS derived_table
WHERE nbr_entries > 1;
```

此语句使用了一个子查询，该子查询根据 GROUP BY 子句统计出每个名字及该名字的计数。虽然 7.7 节已经讲过 GROUP BY 和 COUNT()，但这里我们还是再走一遍。此子查询查出 name_first 和 name_last，并对它们进行分组，以使相同的姓名被归在一组。于是，我们就可以对每一组进行计数了。我们还给 COUNT(*) 分配了一个别名（nbr_entries），以便在语句的其他位置能引用该计数。

然后再来看主 SQL 语句。它用 WHERE 子句筛选出子查询中姓名重复的行（即 nbr_entries 大于 1 的行）。这些就是重复的姓名。于是，整个语句的意思就是，在 humans 表中找出重复的姓名，然后将其输入临时表，并且每个姓名在临时表中只占一行。

现在我们在临时表中已经知道了可能重复注册的人，接着就可以更新 humans 表，把他们标识出来了：

```
UPDATE humans, possible_duplicates
SET possible_duplicate = 1
WHERE name_first = name_1
AND name_last = name_2;
```

此语句将把 humans 表中符合 possible_duplicates 的行更新成 possible_duplicate = 1。标记好之后，我们就可以给这些用户发邮件，告诉他们有重复的姓名，问他们是否重复注册。如果有人回复说是，那么就可以对该用户的信息进行一些整合（例如，增加一列，以存放该用户的第二个电子邮件地址），并删掉重复的行。而那个临时表会自然地在退出 MySQL 客户端时被删掉。

8.2 删除数据

大部分数据库都需要有删除数据的操作。该操作可以通过 DELETE 语句来完成。而之前我们已经多次提过，MySQL 没有 UNDELETE 或 UNDO 命令供你恢复被删数据。恢复数据只能通过备份，而且你也应该做好备份，不过恢复备份是很缓慢、很麻烦的。如果你用的是 InnoDB，那么可以用事务来集合 SQL 语句，并对事务中执行过的删除操作进行回滚。但如果在事务中你把删除都提交了，那是无法回滚的，你就只能找备份或其他麻烦的方法来恢复了。因此，删除数据要谨慎。

DELETE 语句与 SELECT 语句相似，可以根据 WHERE 子句来指定要删除的行。而且，你也应该总是加上 WHERE 子句，除非你真的想把表中的所有行都删掉。另外，你可以带上 ORDER BY 子句，以指定删除的顺序，也可以带上 LIMIT 子句，按行数来限定删除量。它的基本语法如下：

```
DELETE FROM table
[WHERE condition]
[ORDER BY column]
[LIMIT row_count];
```

WHERE、ORDER BY 和 LIMIT 都是用方括号包围的，代表它们都是可选的。除此之外，还有其他选项可实现一次删除多个表的数据，或根据其他表删除某个表的数据。不过，我们现在先看最简单的这种。

假设在发邮件给那些疑似重复注册的用户之后，有人回复承认了。并且，这个人是来自俄罗斯的 Elena Bokova，她希望我们帮她删掉那个旧的雅虎邮件地址。那么，我们只需执行下面这条语句（但实际上我们不执行它）：

```
DELETE FROM humans
WHERE name_first = 'Elena'
AND name_last = 'Bokova'
AND email_address LIKE '%yahoo.com';
```

此 SQL 语句会删掉所有符合 WHERE 中条件的行。注意，为了找到用户所说的那个电子邮件地址，我们使用了 LIKE 和 %，以匹配所有以 yahoo.com 结尾的值。

这个语句是可行的，不过，我们还得删掉 prize_winners 中相关的记录。所以其实我们应该在删除 humans 表中相应行之前，先把 human_id 记下来。这就是为什么我刚才说不要执行该语句。但是，这样先用一条语句获取 human_id，然后用另一条语句删除 humans 中的记录，最后再用一条语句删除 prize_winners 中的记录，是很繁琐的。更好的做法是，在 DELETE 中包含这两个表，一次就删除两个表的相关记录。具体怎么做，下一节会介绍。

一次删除多个表的数据

一个表的数据依赖另一个表的数据是很常见的。如果用 DELETE 从一个表中删掉了一行被另一个表参考的数据，那么就会产生孤立数据。可以再写一条 DELETE 语句来把另一个表的相关行给删掉，但更好的做法是在同一条语句中删除两个表中的行。尤其是当要删的行很多时，更应该采用后者。

DELETE 的多表删除语法如下：

```
DELETE FROM table[, table]
USING table[, . . . ]
[WHERE condition];
```

你需要在 FROM 子句中，以逗号分隔的方式，列出相关的表。然后，在 USING 子句中，声明这些表是如何连接的（例如，用 human_id 连接）。最后的 WHERE 子句是可选的。和 UPDATE 的多行更新一样，有多个表时，不能使用 ORDER BY 和 LIMIT。这种语法是挺棘手的，光看上面的模板可能不太明显，需要用例子来说明。

上一节的最后一个例子讲到删除两个表中相关的行。具体来说，就是把使用雅虎邮箱的用户 Elena Bokova 从 humans 和 prize_winners 两个表中删除。想要用一条语句完成此任务，可以这样做：

```
DELETE FROM humans, prize_winners
USING humans JOIN prize_winners
WHERE name_first = 'Elena'
AND name_last = 'Bokova'
AND email_address LIKE '%yahoo.com'
AND humans.human_id = prize_winners.human_id;
```

此 DELETE 与其他数据操作的语句很像（如 SELECT、UPDATE）。不过，它却有一点出乎意料，甚至令人困惑的地方。我们在 FROM 子句列出想删的表后，又在 USING 子句中再次列出它

们，并声明它们的连接方式。很明显，FROM 是用于说明要删哪些表的。如果没有在 FROM 中包含 prize_winners，那么该表中相关的行将不会被删除。

除了以上这些，DELETE 还有其他选项和写法。不过，就现阶段来说，本章所介绍的内容已经能够帮助你完成绝大部分的 MySQL 和 MariaDB 的开发和管理任务了。

8.3 小结

UPDATE 和 DELETE 适用于修改表中的数据。而且，对于管理 MySQL 和 MariaDB 来说，它们是必不可少的。它们的写法多样，能助你轻松修改数据。你也可以使用它们构建出复杂的 SQL 语句，精确地更新或删除指定的数据。不过，它们有时令人迷惑。所以，你应时刻保持警觉，并努力掌握好它们的用法。

如果你在使用 UPDATE 和 DELETE 时提心吊胆，那就对了。一条 UPDATE 语句可以修改表中所有的行，一条 DELETE 语句就可以删除表中所有的行。对于大型数据库来说，那就可能导致数千行数据在一秒钟之内被更改或删除。所以，我们才需要做好备份。无论何时，在使用这两条语句之前，都应该先检查清楚它们是否无误。尤其是对于初学者来说，最好先用 CREATE TABLE...SELECT 把表复制一份，再做修改或删除。5.2 节介绍过这种用法。使用这种方法，即使改错了数据，也还能用备份出来的数据恢复回去。

多加练习 UPDATE 和 DELETE 吧。如果你使用出错的话，结果不仅影响到你，还会影响到其他与你使用同一个数据库的人。所以，对于下一节的习题，你应比以往更认真地完成它们。

8.4 习题

以下是关于 UPDATE 和 DELETE 的习题。如果你还未从 MySQL 资源站（http://mysqlresources.com/files）下载 rookery 和 birdwatchers 库，请先去下载。它们所提供的数据量应该足以让你完成以下题目。

(1) 用 CREATE TABLE...SELECT 语句（详见 5.2 节）复制 humans 表和 prize_winners 表，并将新表分别命名为 humans_copy 和 prize_winners_copy。复制完后，用 SELECT 语句查出两个表所有的行。结果应该与原表一样才对。

(2) 做完上题之后，用 SELECT 语句查出 humans 表中所有奥地利的会员。奥地利的 country_id 是 au，你会用到 WHERE 子句。出现问题的话，排查并修正，直至问题解决。

接着，抽出刚才 SELECT 中的 WHERE 子句，构建一个 UPDATE，将所有奥地利的会员改成高级会员。并且，在同一 UPDATE 语句中，将 membership_expiration 改成一年之后（从你执行语句的时刻算起）。这里你需要在 DATE_ADD() 中使用 CURDATE()。而 CURDATE() 是不需要参数的，即其括号中不应有任何内容。第 11 章将详细介绍这两个函数。如果你不懂如何将两个函数组合在一起，也可以直接输入日期（如用 '2014-11-03' 表示 2014 年 11 月 3 日，记得加上引号）。更新数据之后，用 SELECT 检查修改是否正确。

(3) 用 DELETE 语句删掉 humans 和 prize_winners 中 Barry Pilson 会员相关的行。我们在讲解删除多个表的数据时解释过怎样做。删完之后，用 SELECT 检查操作是否无误。

(4) 用 DELETE 删掉 humans 所有的数据，接着再删 prize_winners 的数据。然后用 SELECT 确认是否两个表都空了。

空了以后，用 INSERT...SELECT（详见 6.3.2 节），将 humans_copy 和 prize_winners_copy 表的数据分别复制到 humans 和 prize_winners 中。

数据都导回来之后，再次用 SELECT 确认数据是否完整。如果是，则用 DROP TABLE 删掉 humans_copy 和 prize_winners_copy 表。第 4 章和第 5 章都提到过 DROP TABLE 语句。无论何时，如果删错了表或行，都可以从 MySQL 资源站找回整个数据库，继续练习。

第 9 章

表连接和子查询

为了让你着眼于基本的语法，此前的大多数例子都刻意使每条 SQL 语句只操作一个表。不过，在实际的数据库开发中，多表查询是很常见的。它的实现方法有很多，其中有些已经在前面的章节中讲过。而本章将会深入讨论以下话题：如何合并多个结果集，如何连接表，以及如何使用子查询得出类似的结果集。

9.1 合并结果集

先看看如何将多个 SQL 语句的结果集合并。有些时候，你只是想将两个不相关的 SELECT 语句的结果合在一起。对于这种情况，可以用 UNION 运算符。它能组合两个 SELECT 语句，得出一个合并的结果集（其实更多的 SELECT 也是可以的，你只需在各个 SELECT 之间写上 UNION 即可）。下面让我们来看看具体的例子。

在 7.7 节中，我们试过统计 birds 表中 Pelecanidae 科（即 Pelican）的鸟种数量。现在假设我们还想求出 Ardeidae 科（即 Heron）的鸟种数量。那很简单：复制原本的 SELECT，然后改改 WHERE 中的条件。而如果你想将这两个结果集合在一起显示，那么可以用 UNION。现在，试试在 mysql 中输入以下语句：

```
SELECT 'Pelecanidae' AS 'Family',
  COUNT(*) AS 'Species'
  FROM birds, bird_families AS families
  WHERE birds.family_id = families.family_id
  AND families.scientific_name = 'Pelecanidae'
UNION
SELECT 'Ardeidae',
  COUNT(*)
  FROM birds, bird_families AS families
  WHERE birds.family_id = families.family_id
```

```
    AND families.scientific_name = 'Ardeidae';

+-------------+---------+
| Family      | Species |
+-------------+---------+
| Pelecanidae |      10 |
| Ardeidae    |     157 |
+-------------+---------+
```

首先需要注意的是，结果集的标题只从第一个 SELECT 语句中获取。再者，这两个 SELECT 的第一个域，都不是取自实际的列内容，而是取自我们给出的文本：'Pelecanidae' 和 'Ardeidae'。在 MySQL 和 MariaDB 中，这是可以的。如果你想创造出一个域，大可以这样写。而且，因为使用 UNION 时，MySQL 只会取第一个 SELECT 的域名作为最终结果集的标题，而之后的 SELECT 的域名都会被忽略，所以我们只在第一个 SELECT 中设置了域的别名。如果什么别名都不起，那么它就直接取 UNION 中第一个 SELECT 的列名。

关于 UNION，还要记住：它只能与 SELECT 语句一起用；这些 SELECT 所查的表可以不同；重复的行在结果集中合并为一列。

合并后的结果集，可以用 ORDER BY 来排序。如果你想单独对某个 SELECT 的结果集排序，可以用括号将该 SELECT 括起来，并加一个 ORDER BY 将其包围（如果是对整个 UNION 排序）。在 ORDER BY 中指定列时，不能在列名的前面带有（原表的）表名（比如 families.scientific_name）。如果该列名有歧义，就需要使用列的别名。为了更好地演示如何在用到 UNION 时使用 ORDER BY，我们来扩展一下刚才的例子：查出 Pelecaniformes 目和 Suliformes 目中各个科的鸟种数量。用以下语句：

```
  SELECT families.scientific_name AS 'Family',
    COUNT(*) AS 'Species'
    FROM birds, bird_families AS families, bird_orders AS orders
    WHERE birds.family_id = families.family_id
    AND families.order_id = orders.order_id
    AND orders.scientific_name = 'Pelecaniformes'
    GROUP BY families.family_id
UNION
  SELECT families.scientific_name, COUNT(*)
    FROM birds, bird_families AS families, bird_orders AS orders
    WHERE birds.family_id = families.family_id
    AND families.order_id = orders.order_id
    AND orders.scientific_name = 'Suliformes'
    GROUP BY families.family_id;

+-------------------+---------+
| Family            | Species |
+-------------------+---------+
| Pelecanidae       |      10 |
| Balaenicipitidae  |       1 |
| Scopidae          |       3 |
| Ardeidae          |     157 |
| Threskiornithidae |      53 |
| Fregatidae        |      13 |
| Sulidae           |      16 |
| Phalacrocoracidae |      61 |
```

```
| Anhingidae        |       8 |
+-------------------+---------+
```

结果中，前五行是 Pelecaniformes 的，剩下的是 Suliformes 的。它们没有按字母排序，而是按每个 SELECT 语句的执行顺序列出，而每个 SELECT 内的顺序，又按 family_id 来排。如果想全部按科名的字母序来排，那就需要在这个结果集的后面加上 ORDER BY。具体来说，就是将该结果集包在括号中，以令 MySQL 将其当成一个表。然后，查询该“表”所有行的所有列，并用 ORDER BY 来指定最终按科名排序。另外，为了避免混淆，我们要给结果集加上目名。如下：

```
  SELECT * FROM
(
    SELECT families.scientific_name AS 'Family',
    COUNT(*) AS 'Species',
    orders.scientific_name AS 'Order'
    FROM birds, bird_families AS families, bird_orders AS orders
    WHERE birds.family_id = families.family_id
    AND families.order_id = orders.order_id
    AND orders.scientific_name = 'Pelecaniformes'
    GROUP BY families.family_id
UNION
  SELECT families.scientific_name, COUNT(*), orders.scientific_name
    FROM birds, bird_families AS families, bird_orders AS orders
    WHERE birds.family_id = families.family_id
    AND families.order_id = orders.order_id
    AND orders.scientific_name = 'Suliformes'
    GROUP BY families.family_id ) AS derived_1
ORDER BY Family;

+-------------------+---------+----------------+
| Family            | Species | Order          |
+-------------------+---------+----------------+
| Anhingidae        |       8 | Suliformes     |
| Ardeidae          |     157 | Pelecaniformes |
| Balaenicipitidae  |       1 | Pelecaniformes |
| Fregatidae        |      13 | Suliformes     |
| Pelecanidae       |      10 | Pelecaniformes |
| Phalacrocoracidae |      61 | Suliformes     |
| Scopidae          |       3 | Pelecaniformes |
| Sulidae           |      16 | Suliformes     |
| Threskiornithidae |      53 | Pelecaniformes |
+-------------------+---------+----------------+
```

以上例子似乎是事倍功半。但有些时候（虽然这种时候不多），UNION 确实是最好、最简单的做法。例如，当你需要从不同的表获取数据时，又或者是尽管发起多条 SQL 语句更简单，但却需要一次就查出的时候，UNION 都能很方便地给出一个整合好的结果集。

除了 UNION，上例用子查询来做也可以达到同样效果，而且更省事。不过其实上例也是一种子查询，因为我们将 UNION 包在括号里了。本章将在稍后介绍子查询。现在我们先看看如何在一条 SQL 语句中连接多个表。

9.2　表连接

我们可以基于列来使用 JOIN 子句连接两个表，并将这个连接放在 SELECT、UPDATE 或 DELETE 中，以做数据的查询、更新或删除操作。例如，在 3.3.3 节中，因为 books 表的 status 列和 status_names 表的 status_id 列都保存了状态的标识号，所以我们可以用这两列来进行连接，查出同一本书在两个表上对应的信息：

```
SELECT book_id, title, status_name
FROM books JOIN status_names
WHERE status = status_id;
```

现在我们用这个例子来回顾一下连接的原理。books 表的 status 列和 status_names 表的 status_id 列都存放着状态的编号。在 books 中，那个编号本身没有含义。但在 status_names 中，它关联着一个状态的表述。因此，通过连接两表，你就能得知书的状态描述了。

有时，你也可以不用 JOIN 子句，直接将需要查询的表以逗号分隔的方式，列在 SQL 语句中的适当位置（如果是 SELECT 语句，就是列在 FROM 子句中），然后，在 WHERE 中写出用以连接的列。我们在前几章用过这种做法。虽然它运行起来没有问题，而且看上去也很简洁，但我还是推荐你用 JOIN 来连接两个表，并特别指定连接条件。当 SQL 语句出错时，将连接条件与筛选条件分开来写，更有利于排查问题。

JOIN 可用于在查询、更新或删除数据时，基于列来进行表连接。它通常写在 SQL 语句中那个指定表名的位置。它可让你在 ON 中写连接条件，而不用在 WHERE 中写。不过，这时你也还是使用等号运算符来指定连接的列。如果列有多对，则用 AND 分隔。而如果每对列中，两列在两表中名字都相同，你甚至可以用 USING 来做。只需在随后的括号中，以逗号分隔的方式，写出两表共有的列即可。使用 USING 时，连接条件中的列必须是在两表中都存在的。另外，为了提高性能，用于连接的列应该加索引。

ON 的写法如下：

```
SELECT book_id, title, status_name
FROM books
JOIN status_names ON(status = status_id);
```

这与之前的那个例子一样，只不过是少了 WHERE。因为有了 ON 来指定连接条件，所以我们已经不需要 WHERE 了。如果要把 books 的 status 列改成与 status_names 的一样，也叫 status_id，那么就可以采用如下写法：

```
SELECT book_id, title, status_name
FROM books
JOIN status_names USING(status_id);
```

这里，我们在 JOIN 子句中使用 USING 关键字，来指定连接条件。

以上只是 JOIN 的多种写法中的两种，它们都演示了如何在 SELECT 中使用 JOIN。在 UPDATE 和 DELETE 中，JOIN 的写法也基本上是这样。接下来的几节会给出一些示例，以说明这三种语句带 JOIN 的各式写法。

9.2.1 基本的表连接查询

假设现在我们想查出生存状况为 Threatened（这是某一类状态）的 Goose 鸟，那么就得写一条 SQL 语句，从 birds 表和 conservation_status 表中获取数据。这两个表所共用的数据是 conservation_status_id 列。尽管没有必要给连接列起相同的名字，但这样做能让人更容易知道怎样连接两表。

在 mysql 中输入以下命令：

```
SELECT common_name, conservation_state
FROM birds
JOIN conservation_status
ON(birds.conservation_status_id = conservation_status.conservation_status_id)
WHERE conservation_category = 'Threatened'
AND common_name LIKE '%Goose%';

+----------------------------+--------------------+
| common_name                | conservation_state |
+----------------------------+--------------------+
| Swan Goose                 | Vulnerable         |
| Lesser White-fronted Goose | Vulnerable         |
| Hawaiian Goose             | Vulnerable         |
| Red-breasted Goose         | Endangered         |
| Blue-winged Goose          | Vulnerable         |
+----------------------------+--------------------+
```

上例用 ON 来指定两表共有的 conservation_status_id 作为连接条件。然后，MySQL 自然就懂得将两表匹配的行牵扯到一起，并从恰当的表中获取 conservation_category 列和 common_name 列。

虽然这么写是正确的，但语句太长了。现在改用 USING，使得 conservation_status_id 只需声明一次。MySQL 会知道怎么做。以下就是将上例改用 USING 运算符的写法：

```
SELECT common_name, conservation_state
FROM birds
JOIN conservation_status
USING(conservation_status_id)
WHERE conservation_category = 'Threatened'
AND common_name LIKE '%Goose%';
```

接着，再改改这条 SQL 语句，加入鸟科信息。为了实现目标，我们要多连接一个表，即 bird_families。同时，试试也查出 Duck：

```
SELECT common_name AS 'Bird',
bird_families.scientific_name AS 'Family', conservation_state AS 'Status'
FROM birds
JOIN conservation_status USING(conservation_status_id)
JOIN bird_families USING(family_id)
WHERE conservation_category = 'Threatened'
AND common_name REGEXP 'Goose|Duck'
ORDER BY Status, Bird;
```

```
+-----------------------------+----------+-----------------------+
| Bird                        | Family   | Status                |
+-----------------------------+----------+-----------------------+
| Laysan Duck                 | Anatidae | Critically Endangered |
| Pink-headed Duck            | Anatidae | Critically Endangered |
| Blue Duck                   | Anatidae | Endangered            |
| Hawaiian Duck               | Anatidae | Endangered            |
| Meller's Duck               | Anatidae | Endangered            |
| Red-breasted Goose          | Anatidae | Endangered            |
| White-headed Duck           | Anatidae | Endangered            |
| White-winged Duck           | Anatidae | Endangered            |
| Blue-winged Goose           | Anatidae | Vulnerable            |
| Hawaiian Goose              | Anatidae | Vulnerable            |
| Lesser White-fronted Goose  | Anatidae | Vulnerable            |
| Long-tailed Duck            | Anatidae | Vulnerable            |
| Philippine Duck             | Anatidae | Vulnerable            |
| Swan Goose                  | Anatidae | Vulnerable            |
| West Indian Whistling-Duck  | Anatidae | Vulnerable            |
| White-headed Steamer-Duck   | Anatidae | Vulnerable            |
+-----------------------------+----------+-----------------------+
```

此例用了两个 JOIN。通常来说，表的先后顺序是无所谓的。比如说，尽管我们将 bird_families 写在 conservation_status 之后，但 MySQL 还是知道与 bird_families 连接的是 birds 表。如果不用 JOIN 的话，就得在 WHERE 中明确指出各表通过什么列来连接，如下所示：

```
SELECT common_name AS 'Bird',
bird_families.scientific_name AS 'Family', conservation_state AS 'Status'
FROM birds, conservation_status, bird_families
WHERE birds.conservation_status_id = conservation_status.conservation_status_id
AND birds.family_id = bird_families.family_id
AND conservation_category = 'Threatened'
AND common_name REGEXP 'Goose|Duck'
ORDER BY Status, Bird;
```

这个 WHERE 混杂了很多东西，令人难以分辨哪个是筛选条件。反观 JOIN 就清晰得多了。

顺便说一下，上面那个含有两个 JOIN 的 SQL 语句还用到了正则表达式（WHERE 中的 REGEXP）来指定查询 Goose 或 Duck。在最后，我们还加了一个 ORDER BY 子句，使结果先按 conservation_state、后按 common_name 排序。

不过，上例列出鸟科的名字是没有意义的，因为所有结果都同属一科。另外，由于可能还存在一些我们想看到的鸟，因为不符合 Goose|Duck 这个条件而没被查出，所以，我们决定改改查询条件，并将结果集按濒危程度从低往高排序。语句修改如下：

```
SELECT common_name AS 'Bird from Anatidae',
conservation_state AS 'Conservation Status'
FROM birds
JOIN conservation_status AS states USING(conservation_status_id)
JOIN bird_families USING(family_id)
WHERE conservation_category = 'Threatened'
AND bird_families.scientific_name = 'Anatidae'
ORDER BY states.conservation_status_id DESC, common_name ASC;
```

```
+---------------------------+----------------------+
| Bird from Anatidae        | Conservation Status  |
+---------------------------+----------------------+
| Auckland Islands Teal     | Vulnerable           |
| Blue-winged Goose         | Vulnerable           |
| Eaton's Pintail           | Vulnerable           |
| Hawaiian Goose            | Vulnerable           |
| Lesser White-fronted Goose | Vulnerable          |
| Long-tailed Duck          | Vulnerable           |
| Marbled Teal              | Vulnerable           |
| Philippine Duck           | Vulnerable           |
| Salvadori's Teal          | Vulnerable           |
| Steller's Eider           | Vulnerable           |
| Swan Goose                | Vulnerable           |
| West Indian Whistling-Duck | Vulnerable          |
| White-headed Steamer-Duck | Vulnerable           |
| Bernier's Teal            | Endangered           |
| Blue Duck                 | Endangered           |
| Brown Teal                | Endangered           |
| Campbell Islands Teal     | Endangered           |
| Hawaiian Duck             | Endangered           |
| Meller's Duck             | Endangered           |
| Red-breasted Goose        | Endangered           |
| Scaly-sided Merganser     | Endangered           |
| White-headed Duck         | Endangered           |
| White-winged Duck         | Endangered           |
| White-winged Scoter       | Endangered           |
| Baer's Pochard            | Critically Endangered |
| Brazilian Merganser       | Critically Endangered |
| Crested Shelduck          | Critically Endangered |
| Laysan Duck               | Critically Endangered |
| Madagascar Pochard        | Critically Endangered |
| Pink-headed Duck          | Critically Endangered |
+---------------------------+----------------------+
```

这里最明显的改变就是，bird_families.scientific_name 列没有了，现在结果中只剩两列。而另一个改变，则是增加了一层接口——给 conservation_status 表起了别名 states，使得在其他地方引用该表时，不用再输入那么长的表名。

最后，ORDER BY 将结果集按 conservation_status_id 来排序，因为在 conservation_status 表中，这些 ID 原本的顺序是从高危到低危的。为了把低危的排在结果集的前头，我们得加一个 DESC。我们保留了按俗名排序，只不过这次写的是实际的列名，而不是别名。这是因为我们起的别名叫 Bird from Anatidae（它们确实全都来自这一科），在 ORDER BY 中输入这串东西太麻烦了。

接着，我们再看一个 JOIN 的简单例子。假设我们想得知有哪些俄罗斯用户（他们的 country_id 是 ru）曾发现过 Scolopacidae 科的鸟（如 Sandpiper 和 Curlew 这样的岸鸟和涉禽），那么可以从 bird_sightings 表查询鸟种发现点的信息。该表还记录了鸟种发现时采集自用户手机的 GPS 坐标信息。输入以下 SQL 语句：

```
SELECT CONCAT(name_first, ' ', name_last) AS Birder,
common_name AS Bird, location_gps AS 'Location of Sighting'
FROM birdwatchers.humans
JOIN birdwatchers.bird_sightings USING(human_id)
JOIN rookery.birds USING(bird_id)
JOIN rookery.bird_families USING(family_id)
WHERE country_id = 'ru'
AND bird_families.scientific_name = 'Scolopacidae'
ORDER BY Birder;

+-------------------+-------------------+-------------------------+
| Birder            | Bird              | Location of Sighting    |
+-------------------+-------------------+-------------------------+
| Anahit Vanetsyan  | Bar-tailed Godwit | 42.81958072; 133.02246094 |
| Elena Bokova      | Eurasian Curlew   | 51.70469364; 58.63746643  |
| Elena Bokova      | Eskimo Curlew     | 66.16051056; -162.7734375 |
| Katerina Smirnova | Eurasian Curlew   | 42.69096856; 130.78185081 |
+-------------------+-------------------+-------------------------+
```

此语句连接了四个表，其中两个来自 birdwatchers 数据库，另外两个来自 rookery 数据库。你可以仔细观察这条 SQL 语句，思考一下为什么要连接这四个表。那是因为，要想得到这个结果集，这四个表都是必需的。顺便说一下，我们还用了 CONCAT() 函数，把会员的姓和名拼接在一起，以组成 Birder 列。

JOIN 还有其他写法，可做各种各样的连接。现在我们就来试试。比如说，查出所有 Egret 及其每种的保护状态。SQL 语句如下：

```
SELECT common_name AS 'Bird',
conservation_state AS 'Status'
FROM birds
LEFT JOIN conservation_status USING(conservation_status_id)
WHERE common_name LIKE '%Egret%'
ORDER BY Status, Bird;

+--------------------+-----------------+
| Bird               | Status          |
+--------------------+-----------------+
| Great Egret        | NULL            |
| Cattle Egret       | Least Concern   |
| Intermediate Egret | Least Concern   |
| Little Egret       | Least Concern   |
| Snowy Egret        | Least Concern   |
| Reddish Egret      | Near Threatened |
| Chinese Egret      | Vulnerable      |
| Slaty Egret        | Vulnerable      |
+--------------------+-----------------+
```

这个 SELECT 与之前那些例子差不多，只是它用的是 LEFT JOIN，而不是 JOIN。这种写法会使数据库查出左表（即 birds）的所有行，而不管其在右表（即 conservation_status）有没有对应的行。因为无法找到对应的行，所以 MySQL 会给 NULL。如以上结果所示，Great Egret 的 Status 为 NULL，因为它在 birds 中的 conservation_status_id 没有信息。总而言之，如果 birds 的 conservation_status_id 是 NULL、空（即 ''），或任何在右表无

法找到对应行的值，那么 Status 都会是 NULL。

因为本例使用的是 LEFT JOIN，所以，birds 中所有俗名带有 Egret 的鸟，包括那些保护状态未知的，都被查出来了。而且，这也提示了我们还需要为哪些鸟填写保护状态。我们可以使用带有同样 LEFT JOIN 的 UPDATE 语句来填写上述信息。下一节将介绍具体做法。

9.2.2　更新已连接的表

如果想用 UPDATE 一次更新多个表，或想以其他表来限制某个表更新哪些行，可以使用 JOIN 子句。在 UPDATE 中使用 JOIN 与在 SELECT 中无异。所以，我们就直接看一些实际的例子吧。就从上一小节的最后那个例子开始。

我们需要用 UPDATE 加上 LEFT JOIN，来找出 birds 表中没填 conservation_status_id 的行。虽然可以一次就更新所有符合条件的行，但还是只针对 Ardeidae 科的吧（即 Heron、Egret 和 Bittern）。首先，执行以下 SELECT 语句，检查一下连接条件与筛选条件：

```
SELECT common_name,
conservation_state
FROM birds
LEFT JOIN conservation_status USING(conservation_status_id)
JOIN bird_families USING(family_id)
WHERE bird_families.scientific_name = 'Ardeidae';
```

如果你用的是从 MySQL 资源站下载的测试数据，那么应该能查出 150 多行。你会看到其中很多行的 common_name 都是空的。那是因为确实有很多鸟种只有学名，没有俗名。同时，这些鸟的 conservation_status_id 也为空。而在那些有俗名的鸟中，也有一些的 conservation_status_id 为空。

现在，给 conservation_status 增加一行，以代表“不明状态”（Unknown）。接着我们把那些 conservation_status_id 为空的行，更新成这种状态。输入以下两条 SQL 语句：

```
INSERT INTO conservation_status (conservation_state)
VALUES('Unknown');

SELECT LAST_INSERT_ID();

+------------------+
| LAST_INSERT_ID() |
+------------------+
|                9 |
+------------------+
```

在第一条 SQL 语句中，我们只为 conservation_state 输入了值，其他列按默认就行了。因为我们接下来就要更新 birds 中 Ardeidae 鸟的 conservation_status_id，所以得先知道 Unknown 状态的 ID。LAST_INSERT_ID() 能返回本连接（即当前连接）最近一条 SQL 语句所生成的 ID 值。利用该 ID 值，我们就可以更新 birds 的 conservation_status_id。SQL 语句如下，ID 按实际填写。

```
UPDATE birds
LEFT JOIN conservation_status USING(conservation_status_id)
```

```
JOIN bird_families USING(family_id)
SET birds.conservation_status_id = 9
WHERE bird_families.scientific_name = 'Ardeidae'
AND conservation_status.conservation_status_id IS NULL;
```

此 UPDATE 语句应该会更新近 100 行。其中的连接与之前的那个查询哪些 Great Egret 没有保护状态的 SELECT 里的一样。但在 WHERE 中我还加了一个 conservation_status.conservation_status_id IS NULL 这样的条件。虽然我们也可以去掉 LEFT JOIN，直接更新 birds 中 conservation_status_id 为 NULL 的行，但这就会忽略了那些找不到对应状态的行（例如，conservation_status_id 为空字符串）。而使用 LEFT JOIN 后，我们就囊括了 birds 中 conservation_status_id 各种可能的取值。不过这就需要按 conservation_status.conservation_status_id IS NULL 来判断，因为如果找不到匹配状态的话，结果集中该列会返回 NULL。

因为 JOIN 在 SELECT 和 UPDATE 中的用法一样，所以在 UPDATE 之前，你可以轻松地先用 SELECT 测试 JOIN 和 WHERE 写得是否正确。确认无误后，再把它放到 UPDATE 中执行。这是更新多表的最佳流程。

9.2.3 从已连接的表中删除数据

看过 SELECT 和 UPDATE 的 JOIN 例子后，现在看看 DELETE。8.2 节已经介绍过一种在 DELETE 中使用 JOIN 的方法。该例中，我们的目的是删除 birdwatchers 库的 humans 表和 prize_winners 表中，拥有 yahoo.com 邮箱的会员 Elena Bokova。当时我们所写的 DELETE 语句确实能达到要求，但它其实有一个潜在问题。现在回头看看那个 SQL 语句。

```
DELETE FROM humans, prize_winners
USING humans JOIN prize_winners
WHERE name_first = 'Elena'
AND name_last = 'Bokova'
AND email_address LIKE '%yahoo.com'
AND humans.human_id = prize_winners.human_id;
```

虽然与之前那些 JOIN 不太一样，但这就是 JOIN 在 DELETE 中的写法。FROM 子句列出需要删除数据的表。USING 子句列出要在 WHERE 子句中用于指定筛选条件的表。USING humans JOIN prize_winners 的意思是，在 WHERE 子句中会使用 humans 和 prize_winners 这两个表的列，来筛选哪些行会被删除。

不要把 USING...JOIN 和 JOIN...USING 搞混了。

按照前面的 DELETE 语句，如果 humans 中有名字和电子邮箱都符合的行，那么 prize_winners 中也必须有相应的 human_id，这样才会在两个表中删除数据。如果 prize_winners 不匹配，则 MySQL 连 humans 也不会删除，并且它不会报错——于是你可能对错误毫无察觉。考虑到这种情况，我们使用 LEFT JOIN：

```
DELETE FROM humans, prize_winners
USING humans LEFT JOIN prize_winners
ON humans.human_id = prize_winners.human_id
WHERE name_first = 'Elena'
AND name_last = 'Bokova'
AND email_address LIKE '%yahoo.com';
```

注意，以上语句使用 LEFT JOIN 和 ON 将 human_id 的连接写在了 USING 中，而不是 WHERE 中。这是必须的，因为若写在 WHERE 中，如果两个表不匹配，则结果集中两表的行都不会出现，导致两表的数据都不会被删除。在 LEFT JOIN 的情况下，符合筛选条件并且两表匹配的行自然会被删除，而那些符合筛选条件，但在 prize_winners 中找不到对应行的 humans 行，也会被删除。这样做可以消除孤立行。

在日常运维中，我们应该偶尔检查一下 prize_winners 中是否有在 humans 中找不到对应记录的行。如果有，就删掉。出现这种情况可能是因为有人要求我们删除账号，但我们删的时候忘了删其他关联的表。对于"prize_winners 有但 humans 无"这种问题，我们可以用 RIGHT JOIN。

```
DELETE FROM prize_winners
USING humans RIGHT JOIN prize_winners
ON humans.human_id = prize_winners.human_id
WHERE humans.human_id IS NULL;
```

此 FROM 子句中，只有 prize_winners 一个表，这是因为我们只想删除该表的数据。使用 DELETE 语句时，如果某些表不需要执行删除操作，就不用在 FROM 中列出这些表。这是一种良好的规范，即使你认为它们不可能被影响到。

因为在 USING 中，我们将 humans 放在 prize_winners 左边，并使用 RIGHT JOIN，所以，即使连接时在左表（humans）中找不到对应行，右表（prize_winners）的数据也会被删除。如果掉转表的位置，则需要换成 LEFT JOIN 才能删除 prize_winners 的数据。

我们还应留意上例中最后那个用 WHERE 来检查 NULL 的子句。就像早前我们看到过的，LEFT JOIN 和 RIGHT JOIN 在找不到对应行时，会返回 NULL 组成的行。因此，可以通过 humans.human_id IS NULL 来找出 prize_winners 的孤立数据。

JOIN 的写法有很多。9.2.1 节所介绍的那些基本语法是最应该好好掌握的，因为它们是最常用的。除此之外，有时你还会用到 LEFT JOIN 和 RIGHT JOIN。接下来，我们会探讨一个相关的话题。它在很多情况下都很适用，那就是子查询。

9.3 子查询

子查询是指一个查询被包含在另一个查询之中，即将一个 SELECT 放在另一条 SQL 语句当中。子查询可以返回一个值、一行数据、多行数据的某一列，或多行数据的多列。它们各被称为标量子查询、列子查询、行子查询和表子查询。本章将一一介绍这几种子查询。

虽然可以用 JOIN（或者有时用 UNION）得到相同效果，但在某些情景中，子查询看起来更清晰。它能将复杂的查询模块化，使语句更容易组织和调试。以下就是两种通用的写法（其实我们在第 8 章也用过子查询）：

```
UPDATE table_1
SET col_5 = 1
WHERE col_id =
  SELECT col_id
  FROM table_2
  WHERE col_1 = value;

SELECT column_a, column_1
FROM table_1
JOIN
  (SELECT column_1, column_2
   FROM table_2
   WHERE column_2 = value) AS derived_table
USING(col_id);
```

第一个例子中的 SELECT 是内部查询，而 UPDATE 则是外部查询（或称主查询）。在第二个例子中，括号里的 SELECT 是内部查询，括号外的是外部查询。包含子查询的外部查询可以是 SELECT、INSERT、UPDATE、DELETE、DO 或者甚至是 SET。不过有个限制：外部查询一般不能查询或修改内部查询所查的表。（但在 FROM 中用子查询时，没有这样的限制。）

这些通用写法可能有点难以理解。其实子查询并没有特别的语法，它只是一种组织 SQL 语句的方法。你只需要考虑以下两点。

第一，如何将子查询置于外部查询之中。举个例子，如果外部查询是 UPDATE，那么可以将子查询放在 WHERE 中，用于筛选哪些行需要更新（就像刚才第一个例子）；也可以把它放到外部 SELECT 的 FROM 中（如第二个例子）。这些位置都是可以的。一个外部查询可以包含多个子查询，但它们一般都被放在 FROM 和 WHERE 中。

第二，子查询所返回的结果是否符合外部查询所需。比如说，在第一个例子中，UPDATE 里的 WHERE 需要子查询返回一个值。如果你写的子查询返回多个值、一行数据或一个表式的结果集，那么 MySQL 会报错。总之，子查询必须返回外部查询所需类型的数据。

再多看一些例子，你就会明白上述两点。如本节开头所提到的，子查询分为标量子查询、列子查询、行子查询和表子查询。以下几小节将通过示例来介绍每一种类型。

9.3.1　标量子查询

最基本的子查询就是返回单个值的标量子查询。它在 WHERE 中与 = 搭配起来特别有用。事实上，只要是接受返回单个值的表达式的地方，都可以放置标量子查询。下面来看一个简单例子。查出 Galliformes 目的鸟科（即 Grouse、Partridge、Quail 和 Turkey）。birds 和 bird_families 的关联可以用 JOIN 轻松实现。现在用标量子查询试试。

```
SELECT scientific_name AS Family
FROM bird_families
WHERE order_id =
  (SELECT order_id
   FROM bird_orders
   WHERE scientific_name = 'Galliformes');
```

```
+----------------+
| Family         |
+----------------+
| Megapodiidae   |
| Cracidae       |
| Numididae      |
| Odontophoridae |
| Phasianidae    |
+----------------+
```

此处的内部查询（子查询）只返回一个值，order_id。它用于补充外部查询的 WHERE 子句。这很简单。我们再看一个例子。

在 9.2.1 节中，有个例子是查询观察到 Scolopacidae 科的俄罗斯用户。现在为了答谢他们使用我们的手机应用来登记发现，我们打算从他们之中抽一个人，给他（她）一年的高级会员权限。这会用到以下 SQL 语句：

```
UPDATE humans
SET membership_type = 'premium',
membership_expiration = DATE_ADD(IFNULL(membership_expiration,
  CURDATE()), INTERVAL 1 YEAR)
WHERE human_id =
  (SELECT human_id
   FROM
     (SELECT human_id, COUNT(*) AS sightings, join_date
      FROM birdwatchers.bird_sightings
      JOIN birdwatchers.humans USING(human_id)
      JOIN rookery.birds USING(bird_id)
      JOIN rookery.bird_families USING(family_id)
      WHERE country_id = 'ru'
      AND bird_families.scientific_name = 'Scolopacidae'
      GROUP BY human_id) AS derived_1
   WHERE sightings > 5
   ORDER BY join_date DESC
   LIMIT 1);
```

这里，最内层的那个查询与我们之前用来查找“观察到 Scolopacidae 科的俄罗斯用户”的语句差不多。区别在于，这次不查用户名，而是查 human_id 和 join_date（会员注册时间）。然后，对 bird_sightings 中符合鸟科条件和国家条件的条目，使用 GROUP BY，将结果集按 human_id 分组，并用 COUNT() 进行计数。这个子查询会返回一个像表一样的结果集，所以，它是表子查询。我们会在后面详细介绍它。

包住这个最内层子查询的那个 SELECT，也是一个子查询。它只查出最内层结果集中 sightings 大于 5 的行，并将新用户排在前面。我们希望给予发现 Curlew 鸟的最新注册的俄罗斯用户一年的高级权限。这个子查询只返回一行一列，它是一个标量子查询。

最后，主查询用这个标量子查询所返回的单个值，定位到那个要被给予高级权限的用户。如果在标量子查询中没有 LIMIT，那么它就会返回多个值——这样就不是标量子查询了。因为外部查询的 WHERE 使用的是 =，它只接受单个值，所以 MySQL 会返回一条这样的错误信息：

```
ERROR 1242 (ER_SUBSELECT_NO_1_ROW)
SQLSTATE = 21000
Message = "Subquery returns more than 1 row"
```

其实所有的子查询都可以用 JOIN 或其他复杂的手法来替换。在某种程度上，这取决于你的编码风格。而我一般就喜欢用子查询，尤其是在 PHP 或 Perl 应用中。这有利于我在几个月或几年以后想要做修改时，识别自己写过的代码。

9.3.2　列子查询

上一小节已讨论过如何在 WHERE 中使用标量子查询。不过，有些时候我们还可能会需要匹配多个值。这种多值匹配的情况，需要用子查询加上 IN，而 IN 通常需要带上一堆以逗号分隔的值。现在我们来看看它。

在上一小节的一个例子中，我们试过用标量子查询来查出 Galliformes 目的所有科。现在假设我们希望结果集中每个科都带有一个该科某一种鸟的俗名，而这个“某一种鸟”，要从每科随机抽取。想实现这种效果，我们得写一个子查询，查出该目的所有科名。输入以下 SQL 语句：

```
SELECT * FROM
  (SELECT common_name AS 'Bird',
   families.scientific_name AS 'Family'
   FROM birds
   JOIN bird_families AS families USING(family_id)
   JOIN bird_orders AS orders USING(order_id)
   WHERE common_name != ''
   AND families.scientific_name IN
     (SELECT DISTINCT families.scientific_name AS 'Family'
      FROM bird_families AS families
      JOIN bird_orders AS orders USING(order_id)
      WHERE orders.scientific_name = 'Galliformes'
      ORDER BY Family)
   ORDER BY RAND()) AS derived_1
GROUP BY (Family);

+------------------------+----------------+
| Bird                   | Family         |
+------------------------+----------------+
| White-crested Guan     | Cracidae       |
| Forsten's Scrubfowl    | Megapodiidae   |
| Helmeted Guineafowl    | Numididae      |
| Mountain Quail         | Odontophoridae |
| Gray-striped Francolin | Phasianidae    |
+------------------------+----------------+
```

这个 SQL 语句有两个子查询，一个套着另一个，然后再被最外层的查询包着。最里层的是一个嵌套子查询。这里所有的子查询都会比其外层的先执行，也就是说，执行第二层的 WHERE 时，可以使用最内层的结果集。对于这里的嵌套子查询来说也一样，它会先于外面的那个子查询执行。此例中，嵌套子查询（即缩进最多的那个）被包在 IN() 中，它查的是 Galliformes 目含有的全部科名。而加在别名 Family 上的 DISTINCT 会对科名进行去重。如

果手动输入科名的话，就会是这样：('Cracidae','Megapodiidae','Numididae','Odontophoridae','Phasianidae')。这个子查询是一个多值子查询，或叫列子查询。

其中，第二层的查询是一个表子查询。它列出最内层子查询提供的鸟科所含的全部鸟种。在这一层上，我们可以对科进行 GROUP BY 操作，获取每个科中的一种鸟。不过，它获取到的都是结果集中每个科的第一行数据，这样每次运行该 SQL 语句，得到的鸟都会是同一种。而我们希望每次随机显示不同的鸟，因此，得先对结果集使用 ORDER BY RAND() 来打乱顺序。最后才用另一个 SELECT 包住它，并对科进行 GROUP BY 操作，获取每个科中的一种鸟。

9.3.3 行子查询

行子查询返回单行数据，供外层查询使用。你可以在 WHERE 中用它来与一组列进行比较，以筛选数据。我们先讲一个例子，然后再深入探讨。假设又有一个观鸟网站关闭了，这是一个东欧的网站。他们把自己的数据库发给我们，里面有一个会员名字的表，还有一个保存了会员所发现的鸟类的表。我们打算将这两个表导入我们的 birdwatchers 库。在导入的过程中，我们发现有些人已是我们的会员——不过这没问题，我们知道如何去重。而对于鸟类发现表，因为会员是有重复的，所以鸟类发现的登记可能也有重复。于是我们得逐条检查，确保只有不重复的才能被导入。看看以下 SQL 语句：

```
INSERT INTO bird_sightings
(bird_id, human_id, time_seen, location_gps)
VALUES
  (SELECT birds.bird_id, humans.human_id,
   date_spotted, gps_coordinates
   FROM
     (SELECT personal_name, family_name, science_name, date_spotted,
      CONCAT(latitude, '; ', longitude) AS gps_coordinates
      FROM eastern_birders
      JOIN eastern_birders_spottings USING(birder_id)
      WHERE
          (personal_name, family_name,
           science_name, CONCAT(latitude, '; ', longitude) )
        NOT IN
          (SELECT name_first, name_last, scientific_name, location_gps
           FROM humans
           JOIN bird_sightings USING(human_id)
           JOIN rookery.birds USING(bird_id) ) ) AS derived_1
   JOIN humans
   ON(personal_name = name_first
      AND family_name = name_last)
   JOIN rookery.birds
   ON(scientific_name = science_name) );
```

这看起来很复杂，很难看懂，或很难写好。我们先来把主要的部分搞清楚。先看看最里层括号的那个嵌套子查询。这里查的是我们自己的数据：会员名字、所发现的鸟种和发现地。这个嵌套子查询被放在另一个子查询的 WHERE 中，那个子查询是一个行子查询。注意，行子查询的列要用括号括起来。因此，这个 WHERE 的意思是，用 eastern_birders 表连接的每一行的这些列，与我们所有的会员名字、所发现的鸟种和发现地进行对比。经对比筛

选后的结果，通过最外层的 INSERT 插入 bird_sightings 表。

这个例子确实很少见，而且好像也不必写得这么复杂。但有些时候像这样的行子查询还是很有用的。简单来说，刚才那个例子要做的是，对于要导入的每一行，如果在我们的库中已经有同一个人在同一地点发现了同一种鸟，则不导入；否则，允许导入。实现这个任务其实也有别的办法，如用一个临时表以及多条 SQL 语句，或使用 Perl 或 PHP 来编写程序。不过，我觉得你还是应该知道一条 SQL 语句也能完成此任务，尤其是在有必要这么写时。

9.3.4 表子查询

子查询可以返回表式的结果集，供外部查询获取数据。把这样的子查询放在 FROM 里，可将它当作表使用。我们把此种情况称作提取表。

使用表子查询需要遵守一些规则。每个提取表都必须有一个别名——只要不与其他表重名即可。起别名时可以使用 AS 关键字。每个提取表内，列名也不能重复。如果提取表中有两个相同的列，那么其中至少有一个得起另外的名字。提取表不可以写成相关子查询，即在提取表中不能引用外部查询的数据。

我们用本章开头的一个 UNION 例子来讲解表子查询。该例分别用两个 SELECT 来统计 Pelecanidae 和 Ardeidae 的鸟种总数，然后，再用 UNION 合并两个结果集。这种写法很笨拙。我们可以用表子查询来改良它。在此子查询中，我们只会选出那两个科的所有鸟的科名。列出这么多重复的科名，看起来好像很蠢，尤其是我们本来就知道它们的科名。但其实，那是为了按科名来分组统计，我们没要 MySQL 将这个子查询的结果显示出来。因为在外部使用了 GROUP BY，所以，最终结果是一个科一条记录。SQL 语句如下：

```
SELECT family AS 'Bird Family',
COUNT(*) AS 'Number of Birds'
FROM
  (SELECT families.scientific_name AS family
   FROM birds
   JOIN bird_families AS families USING(family_id)
   WHERE families.scientific_name IN('Pelecanidae','Ardeidae')) AS derived_1
GROUP BY family;

+-------------+-----------------+
| Bird Family | Number of Birds |
+-------------+-----------------+
| Ardeidae    |             157 |
| Pelecanidae |              10 |
+-------------+-----------------+
```

这样统计两个科的总数，比用 UNION 好得多。要想查其他科，只需在子查询的 WHERE 中添加科名，而不再需要为每个科编写一条 SELECT。

从这个例子可以看出，FROM 中的表子查询，用起来就和普通的表一样。你甚至可以给它起个别名（如 derived_1），就像给表起别名那样。外部查询的 GROUP BY 将该表子查询按 family（子查询中 scientific_name 的别名）分组。另外，在外部 SELECT 中，我们也再次用到了 family。如果表子查询中的列有别名，则外部必须使用该别名来引用该列，原列名在外部将不再可用。

9.3.5　子查询的性能考虑

如果子查询组合得不好，就会有性能问题。在 WHERE 的 IN() 中使用子查询可能会导致性能损失。如果出现这种情况，可以试试用多个 column = value 对来代替，或改用 JOIN，并用 BENCHMARK() 函数比较一下两种写法的性能。你也可以到 Oracle 的网站（http://dev.mysql.com/doc/refman/5.6/en/optimizing-subqueries.html），参考一些子查询的性能优化提示。

9.4　小结

很多开发者都倾向于使用子查询，我也如此。子查询便于组合和拆解，有助于解决问题。如果你使用的是有着庞大访问量的大型数据库，那么子查询就不太适合了，因为可能会有性能问题。而小型数据库则没什么问题。你得学会使用子查询和不使用子查询（例如，用 JOIN 来代替），这样就无惧任何状况。因为你无法预计自己的下一个老板或下一班队友喜欢哪一种写法，所以，两种都会才比较好。

学习 JOIN 是无可避免的。很少有人不用 JOIN。本章几乎所有的子查询示例都显示出，即使你喜欢子查询，但 JOIN 依然有用。而 UNION 应该是很少用到的。学着精通 JOIN，不要对它有抵触，多练习含有 JOIN 的语句，这样才能更好地掌握它。

9.5　习题

以下习题会让你用 JOIN 来连接表或编写子查询。在解题的过程中，多思考一下多个表的数据是怎么走到一起的。试着将每个表幻想成一张纸，纸上是一行行记录，考虑怎样在桌上放置这些纸才能按它们的关系查出数据。你可以用左手食指指着左边的一张纸上的一条记录，然后用右手食指指着右边的另一张纸上的记录。于是，两条记录就连接在一起了。而你两手所指向的那一列，就是连接点。做习题时，想着这个画面，大声说出你在连接什么，你让 MySQL 做什么操作。这样会帮助你更好地理解表连接和子查询。

(1) birdwatchers 库中有个 bird_sightings 表，用来记录会员在野外发现了什么鸟。假设我们举办一场比赛，看看哪个会员发现了最多 Galliformes 目的鸟，并予以奖励。计分规则是每发现一次，得一分。
你需要构建一条 SQL 语句，统计每个会员的发现数。要求结果集显示两列：一列是 human_id，别名为 Birder；另一列是发现数，别名为 Entries。你可以将 bird_sightings 与 birds、bird_families 和 bird_orders 连接起来，以完成此任务。注意，它们不在同一个数据库中。另外，你还需要使用 COUNT() 函数和 GROUP BY 子句。整条语句只能用 JOIN，不得用子查询。最终结果集类似于如下形式：

```
+--------+---------+
| Birder | Entries |
+--------+---------+
|     19 |       1 |
|     28 |       5 |
+--------+---------+
```

做完上题，修改 SQL 语句，以连接 humans 表。拼接出会员的姓名（使用 CONCAT()，姓和名之间要有空格），取代 human_id。别名不变。要求最终结果如下（人名和分数可能不同）。

```
+--------------+--------+
| Birder       | Points |
+--------------+--------+
| Elena Bokova |      4 |
| Marie Dyer   |      8 |
+--------------+--------+
```

(2) 上题已查出每个会员发现了多少次 Galliformes。现在，来些更刺激的。不要每次发现就计分，而是每一科只有一种鸟能被计分，即发现同一种鸟多次，也不会加分。并且，无论一科发现了多少种鸟，也只计一分。我们会让会员根据指南去栖息地找鸟。这样应该更好玩。

为了适应比赛规则的变更，你得修改上题最后的那条 SQL 语句。首先，在外部查询的列前面加上 DISTINCT，并去掉 CONCAT() 和 GROUP BY。改完后，运行一遍，检查是否有误。在正确的结果中，应该会出现某些会员有多行记录。接着，将整条 SQL 语句置于另一条 SQL 语句中，使其成为子查询。而在那个新的外部查询中才用 CONCAT()，并用 GROUP BY 对会员和科进行分组统计。结果大概会是如下形式。

```
+--------------+--------+
| Birder       | Points |
+--------------+--------+
| Elena Bokova |      1 |
| Marie Dyer   |      5 |
+--------------+--------+
```

(3) Galliformes 目有五个科。根据以上的比赛规则，满分应该是 5。现在再改改 SQL 语句，要求只列出 5 分会员。将刚才的 SQL 语句包在另一条 SQL 语句之中，造出嵌套子查询。执行整条 SQL 语句的结果如下。

```
+------------+--------+
| Birder     | Points |
+------------+--------+
| Marie Dyer |      5 |
+------------+--------+
```

第四部分

内置函数

MySQL 有很多内置函数可助你对列中的数据进行操作。有了它们，你就可以格式化数据，提取文本，或者创建搜索表达式。不过，函数本身是不更改表中数据的，其功能仅仅是返回一个新的值。然而，如果在某些 SQL 语句（如 `UPDATE`）中使用得当，我们也是可以用它们来修改数据的。顺便说一下，使用函数时，参数可以是普通的文本或数字，不一定是列。

函数主要分成三类：字符串函数；日期和时间函数；数字和算术函数。字符串函数与文本的转换和格式化相关，当然也包括从列中查找和提取文本。具体内容会在第 10 章讲解。

日期和时间函数在第 11 章讲解。它们可以格式化日期和时间值，以及从某个日期或时间值中抽取内容，甚至还可以从系统获取日期和时间值，以用于插入或更新数据。

数字和算术函数用于数学计算或统计，会在第 12 章讲解。

这三章不会介绍全部的函数，而只会提及各类函数中最常用以及较有用的那些。因为它们也属于学习和开发 MySQL 与 MariaDB 的一部分，所以你应努力掌握它们。

第10章

字符串函数

所谓字符串，就是一个包含字母、数字或其他字符（例如 &、$）的值。一般人们不会把字符串中的数字当成数。而什么时候该用字符串来保存数字，则要看上下文环境，以及该数字所代表的意义。例如，美国的邮编全是由数字组成的，但不能用整数类型存储它们。因为如果用整数类型，02138 就会变成 2138，所以在这种情况下我们只能用字符串。

为了方便处理字符串，MySQL 提供了很多相关的内置函数，可用于格式化字符串，在 `WHERE` 中构造更合适的表达式，或者抽取和修改字符串和列中的内容。所以，这一章就来介绍一些字符串函数。我会将它们按功能分类列出，并提供一些例子，以展示它们的用法。

函数的基本使用方法

使用函数时，有几点需要注意。字符串函数有自己的一些使用规则。其中有些规则会因服务器的设定而异。

- 函数的基本语法，就是关键字后紧接着用括号括起来的参数。注意，与 SQL 运算符（如 `WHERE` 中的 `IN ()`）不同的是，函数关键字与括号之间不能有空格。
- 有些函数不需要参数（例如返回当前日期和时间的 `NOW()`），而其他函数则需要一定数量的参数。各参数一般以逗号分隔，参数的取值可以是其他函数的返回值（函数可以嵌套）。
- 文本作为参数时，需要使用引号。
- 列作为参数时，不要用引号——否则列名会被当成文本。如果列名是保留字或含有可能引起问题的字符，可用反引号标示列名。

- 如果字符串函数返回的值过长（即返回了太多的字符串），超出了系统限制（max_allowed_packet 选项），MySQL 就会返回 NULL。
- 有些参数用于指定字符串中字符的位置。字符串第一个字符的位置是 1，不是 0。当需要从后往前数时（有些函数允许这么做），最后一个字符的位置是 −1。
- 有些参数用于表示字符串长度。如果用到小数，MySQL 就会将其四舍五入为最接近的整数。

10.1 格式化字符串

有些字符串函数可用于格式化或重组文本，以使其显示成更好的形式。所以，你可以将数据按原样保存，或分在多列或多个地方中，然后在获取数据时，用一些字符串函数让它们按你想要的格式展示。

例如在 humans 表中，为了便于按姓或按名来排序，我们将一个人的称谓、名和姓分成三列保存，而当需要完整显示时，用函数将它们拼凑起来。下面我们就来看看这是如何做到的。

10.1.1 拼接字符串

CONCAT() 可以把多列的内容粘在一起，或给某列数据追加内容。它可能是最常用的字符串函数——前面有些例子就已经用过。它的使用方法，就是在括号中，以逗号分隔的方式，写出你想组合的字符串、列或其他元素。

现在看看如何在 SELECT 中使用 CONCAT()。假设我们想得知哪些用户发现了哪些鸟，可以使用如下语句：

```
SELECT CONCAT(formal_title, '. ', name_first, SPACE(1), name_last) AS Birder,
CONCAT(common_name, ' - ', birds.scientific_name) AS Bird,
time_seen AS 'When Spotted'
FROM birdwatchers.bird_sightings
JOIN birdwatchers.humans USING(human_id)
JOIN rookery.birds USING(bird_id)
GROUP BY human_id DESC
LIMIT 4;

+----------------------+---------------------------------+--------------------+
| Birder               | Bird                            | When Spotted       |
+----------------------+---------------------------------+--------------------+
| Ms. Marie Dyer       | Red-billed Curassow - Crax blu...| 2013-10-02 07:39:44|
| Ms. Anahit Vanetsyan | Bar-tailed Godwit - Limosa lap...| 2013-10-01 05:40:00|
| Ms. Katerina Smirnova| Eurasian Curlew - Numenius arq...| 2013-10-01 07:06:46|
| Ms. Elena Bokova     | Eskimo Curlew - Numenius borea...| 2013-10-01 05:09:27|
+----------------------+---------------------------------+--------------------+
```

结果集中的第一列，不是来自表中单个的列，而是用 CONCAT() 将观鸟者的称谓和姓名合成而得的。因为我们保存的称谓不带句点，所以还得另外将句点拼进去。而第二列则是由鸟

的俗名和学名拼接而成，两个名字中间还加了空格和横线。

如果没有 CONCAT()，我们就只能把分开的列合成一列来存储了，比如说，将俗名和学名合成一列，即使它们本该分开。分成多列能使数据库更高效、更具扩展性。当需要合成一个域来显示时，用 CONCAT() 即可。

还有一个不太常用的拼接函数，即 CONCAT_WS()。它可以在拼接列时，在列之间加插指定的分隔符。我们要在第一个参数处写明想要使用的分隔符，而剩下的参数就是被拼接的内容。当需要生成符合其他程序接口的数据时，此函数会很有用。

举个例子，我们现在打算给每个高级用户发一个绣有 Rookery 字样的绣花徽章。其中，配送的事情会交由一家广告和营销公司来处理。该公司需要我们提供一份包含会员姓名和地址的文本文件，并且，每行的每个值要用竖线分隔。于是，我们这么做：

```
mysql -p --skip-column-names -e \
"SELECT CONCAT_WS('|', formal_title, name_first, name_last,
street_address, city, state_province, postal_code, country_id)
FROM birdwatchers.humans WHERE membership_type = 'premium'
AND membership_expiration > CURDATE();" > rookery_patch_mailinglist.txt
```

此例用到了几个 mysql 命令的选项。--skip-column-names 会让 MySQL 不显示列名——因为我们只需要数据部分。-e 的意思是执行后面引号中的内容。于是我们就在它后面写上 SQL 语句，并使用双引号。语句中，CONCAT_WS() 的第一个参数，就是公司所要求的竖线。剩下的参数，就是要串起来的列。双引号闭合之后，我们用了 > 来将结果重定向到一个文本文件中，以便接下来发给该公司。不过这条语句有个潜在的问题：如果某列的值为 NULL，则 CONCAT_WS() 不会输出该列，也不会显示紧贴的两个竖线，来提示你有 NULL 值。这种情况大概如下：

```
Ms|Rusty|Osborne|ch
Ms|Elena|Bokova|ru
```

虽然我们要求返回八列，但这两个会员只输出了四列。如果这两行混在了数千行数据之中，那将会很难被发现，并在之后的处理中导致出错。虽然看起来很笨拙，但我们可以使用 IFNULL() 函数，指定遇到 NULL 时，改为显示其他内容（例如 unknown 或空格）。以下是用 IFNULL() 修改上例的样子：

```
mysql -p --skip-column-names -e \
"SELECT CONCAT_WS('|', IFNULL(formal_title, ' '), IFNULL(name_first, ' '),
IFNULL(name_last, ' '), IFNULL(street_address, ' '),
IFNULL(city, ' '), IFNULL(state_province, ' '),
IFNULL(postal_code, ' '), IFNULL(country_id, ' '))
FROM birdwatchers.humans WHERE membership_type = 'premium'
AND membership_expiration > CURDATE();" > rookery_patch_mailinglist.txt
```

写起来很麻烦、很啰嗦，不过也没什么，反正 MySQL 能运行。于是，结果变成：

```
Ms|Rusty|Osborne| | | | |ch
Ms|Elena|Bokova| | | | |ru
```

这样检查起来就容易多了。它能被营销公司正常导入，而公司在发现有遗漏的信息时，可再联系我们补齐。那时他们只需更新漏掉的那些，而不必重新导入。

10.1.2 设置大小写和引号

有时，你或许会想将某列的文本全部转换成小写，或全部转换成大写。可以用 LOWER() 和 UPPER()，它们又可以分别写成 LCASE() 和 UCASE()。如以下例子，第一列就被转换成了小写，第二列被转换成了大写。

```
SELECT LCASE(common_name) AS Species,
UCASE(bird_families.scientific_name) AS Family
FROM birds
JOIN bird_families USING(family_id)
WHERE common_name LIKE '%Wren%'
ORDER BY Species
LIMIT 5;

+-------------------------+---------------+
| Species                 | Family        |
+-------------------------+---------------+
| apolinar's wren         | TROGLODYTIDAE |
| band-backed wren        | TROGLODYTIDAE |
| banded wren             | TROGLODYTIDAE |
| bar-winged wood-wren    | TROGLODYTIDAE |
| bar-winged wren-babbler | TIMALIIDAE    |
+-------------------------+---------------+
```

QUOTE() 函数接受字符串输入，然后将其用单引号包围，再输出。此外，它还会对某些字符进行转换，使输出内容作为 SQL 语句或其他编程语言的输入时，不会造成问题。会转换的字符包括：单引号、反斜杠、空（零）字节，以及 Ctrl-Z 字符。QUOTE() 会在这些字符前加上反斜杠，以避免它们被当成中断的标志，或过早地闭合单引号。

看看以下查询以 Prince 或 Princess 命名的鸟的代码：

```
SELECT QUOTE(common_name)
FROM birds
WHERE common_name LIKE "%Prince%"
ORDER BY common_name;

+---------------------------------+
| QUOTE(common_name)              |
+---------------------------------+
| 'Prince Henry\'s Laughingthrush' |
| 'Prince Ruspoli\'s Turaco'      |
| 'Princess Parrot'               |
+---------------------------------+
```

因为使用了 QUOTE() 函数，所以结果中的字符串都被单引号包了起来，而且字符串中的单引号都加上了反斜杠。这就避免了传递到其他程序时被识别错的可能。

10.1.3 修剪和补充字符串

网站接受公众输入数据时，需要考虑到，有些人是很粗心的。他们可能会在文本的前后输入空格。想去除列中文本开头或结尾的空格，有几个函数可供选择：LTRIM() 可去掉开头的空格，RTRIM() 可去掉结尾的空格，而 TRIM() 则更厉害，能一下子去掉两头的空格。

你可以用这些函数配合 UPDATE 语句，对数据进行整顿。下面就来看看例子，用 LTRIM() 和 RTRIM() 去除开头和结尾的空格。

```
UPDATE humans
SET name_first = LTRIM(name_first),
name_last = LTRIM(name_last);

UPDATE humans
SET name_first = RTRIM(name_first),
name_last = RTRIM(name_last);
```

此例先用第一条 UPDATE 去掉开头的空格，再用第二条去掉结尾的空格。注意，来源列和目标列都是同一列，但新值已经被去掉空格。我们也可以将两句合成一句：

```
UPDATE humans
SET name_first = LTRIM( RTRIM(name_last) ),
name_last = LTRIM( RTRIM(name_last) );
```

你可以像这样将函数组合在一起，一次就造成更大的变化。不过，如果想去掉两头的空格，直接用 TRIM() 应该更好，如下：

```
UPDATE humans
SET name_first = TRIM(name_first),
name_last = TRIM(name_last);
```

TRIM() 还有很多选项。你可以指定要移除的字符，而不只是空格。例如，假设我们又收到了来自另一个观鸟俱乐部的一份数据，就像 9.3.3 节中一样。不过，在这次所给的表中，鸟种的学名用双引号包起来了。如果想将这些数据导入 bird_sightings 表的话，只需在上次那条 SQL 语句的基础上，加上 TRIM() 即可。以下代码片段是两表的连接方式，我们就是要修改这里：

```
...
JOIN rookery.birds
ON(scientific_name = TRIM(BOTH '"' FROM science_name) ) );
```

可能不太容易看出，但我们确实使用单引号指明了要去掉的字符（双引号）。其中，BOTH 不是必需的，因为本身默认就是 BOTH，所以之前用 TRIM() 时我都没写。如果只想去掉某一端的字符，可以指定 LEADING 或 TRAILING，使 TRIM() 实现 LTRIM() 或 RTRIM() 的效果。而默认被去掉的字符，如之前所见，就是空格。

当需要在 Web 表单或类似的地方展示数据时，你或许会想填充一些像句点之类的字符。例如，对于长度不定的 VARCHAR 列，用可见的字符来填充，让用户看出长度的极限。实现填充的函数有 LPAD() 和 RPAD()，还有 SPACE()，它以空格进行填充。

```
SELECT CONCAT(RPAD(common_name, 20, '.' ),
RPAD(Families.scientific_name, 15, '.'),
Orders.scientific_name) AS Birds
FROM birds
JOIN bird_families AS Families USING(family_id)
JOIN bird_orders AS Orders
WHERE common_name != ''
AND Orders.scientific_name = 'Ciconiiformes'
```

```
ORDER BY common_name LIMIT 3;

+-----------------------------------------------+
| Birds                                         |
+-----------------------------------------------+
| Abbott's Babbler....Pellorneidae...Ciconiiformes |
| Abbott's Booby......Sulidae........Ciconiiformes |
| Abbott's Starling...Sturnidae......Ciconiiformes |
+-----------------------------------------------+
```

注意上例是如何使科和目在垂直方向上对齐的。方法就是，用 RPAD() 把每个值都填充至最大长度。这里，第一个参数是所读取的列，第二个参数是所需的最终结果的总长度，而第三个参数是句点，这就使得原文本长度不足时，会以句点来填充。因为 MySQL 使用的是等宽字体，所以出来的结果会很整齐。你也可以用空格填充，结果也一样会是对齐的。在网页上显示的话，则要用 （不换行空格）来填充。

10.2 抽取文本

有些函数可用于从文本中抽取内容。使用时，需要指定从哪里开始抽（起始位置），以及抽多长。这类函数有四个：LEFT()、MID()、RIGHT() 和 SUBSTRING()。此外，SUBSTRING_INDEX() 也算与此有点关系。下面我会逐一介绍它们。

先看看 LEFT()、MID() 和 RIGHT()。假设我们的营销代理有一个叫 prospects 的表，里面含有一些观鸟者的数据。每个观鸟者的称谓、姓和名都合并在 prospect_name 这一列中，而电子邮箱则由另一列存储。prospect_name 是定长的 CHAR(54)。营销代理说，该列的前 4 个字符是称谓，接着的 25 个字符是名，最后的 25 个字符是姓。其中，称谓只有 Mr. 或 Ms.，并在后面再加一个空格，即用 4 个字符来表示称谓。但对于我们来说，只需抽取前两个字符放到我们的表中即可。现在，先拿四条记录来看看数据的大概样子：

```
SELECT prospect_name
FROM prospects LIMIT 4;

+------------------------------------------------------+
| prospect_name                                        |
+------------------------------------------------------+
| Ms. Caryn-Amy                Rose                    |
| Mr. Colin                    Charles                 |
| Mr. Kenneth                  Dyer                    |
| Ms. Sveta                    Smirnova                |
+------------------------------------------------------+
```

如你所见，称谓、姓和名的长度都是固定的。一般来说，MySQL 保存 CHAR 时，会将结尾的空格去掉。而给该表录入数据的人，应该是使用了 SET sql_mode = 'PAD_CHAR_TO_FULL_LENGTH';，强制数据库按 4、25、25 的格式保存这些文本。

我们可以使用 INSERT INTO…SELECT 以及一些函数，将这些数据进行拆分和抽取，然后导入我们新建的 membership_prospects 表。在真正 INSERT 之前，先用 SELECT 验证函数的使用是否无误：

```
SELECT LEFT(prospect_name, 2) AS title,
MID(prospect_name, 5, 25) AS first_name,
RIGHT(prospect_name, 25) AS last_name
FROM prospects LIMIT 4;

+-------+-------------------------+-------------------------+
| title | first_name              | last_name               |
+-------+-------------------------+-------------------------+
| Ms    | Caryn-Amy               | Rose                    |
| Mr    | Kenneth                 | Dyer                    |
| Mr    | Colin                   | Charles                 |
| Ms    | Sveta                   | Smirnova                |
+-------+-------------------------+-------------------------+
```

由此例可见，LEFT() 抽取数据时，起始位置是第一个字符，而参数中的数字（即 2）是想要抽取的长度。RIGHT() 也类似，不过它是从右往左数的。而 MID() 就有点不同，它可以指定起始位置（例如我们就指定了从第 5 个字符开始）和抽取长度。

SUBSTRING() 与 MID() 相同，因此它们的语法也一模一样。默认情况下，如果没有指定抽取长度，那么它就会抽取从起始位置直至末尾的内容，效果就像 LEFT()。而如果 SUBSTRING() 和 MID() 的第二个参数是负数，则会从右往左抽取该数字所表示的绝对值的长度，效果就像 RIGHT()。

因为 SUBSTRING() 的不同写法能带来不同效果，所以我们可以用它来完成上面的需求。

```
SELECT SUBSTRING(prospect_name, 1, 2) AS title,
SUBSTRING(prospect_name FROM 5 FOR 25) AS first_name,
SUBSTRING(prospect_name, -25) AS last_name
FROM prospects LIMIT 3;
```

此例展示了 SUBSTRING() 的三种用法。

```
SUBSTRING(prospect_name, 1, 2) AS title
```

这种已见过：三个参数分别指定文本的列、起始位置和抽取长度。

```
SUBSTRING(prospect_name FROM 5 FOR 25) AS first_name
```

这是冗长写法。此处 5 代表起始位置，25 代表抽取长度。

```
SUBSTRING(prospect_name, -25) AS last_name
```

将起始位置指定为 -25，但因为没有指定抽取长度，所以 MySQL 抽取从右往左数的 25 个字符。

到底该用哪种，其实随你选择。

SUBSTRING_INDEX() 与 SUBSTRING() 相似，但它不是抽取“哪些字符”，而是“哪些元素”，其中，元素是由特定的分隔符区分的。例如，假设 prospect_name 的结构有变，现在它里面的称谓、名和姓不是定长的，而是用竖线拼接在一起（虽然看起来很怪，但确实可能有人这么保存数据）。于是，我们就将竖线当成分隔符，来拆分这一列（第一和第三行的 SUBSTRING_INDEX() 很好理解，而第二行的就有点复杂了）：

```
SELECT SUBSTRING_INDEX(prospect_name, '|', 1) AS title,
SUBSTRING_INDEX( SUBSTRING_INDEX(prospect_name, '|', 2), '|', -1) AS first_name,
SUBSTRING_INDEX(prospect_name, '|', -1) AS last_name
FROM prospects WHERE prospect_id = 7;
```

SUBSTRING_INDEX() 的第二个参数，用于指定分隔符。于是，我们写上了 '|'。第三个参数用于指定抽取多少个元素。例如，在第三行中，因为是 -1，所以就是从右往左抽取一个。而在第二行中，一个 SUBSTRING_INDEX() 套着另一个，这样，内部的 SUBSTRING_INDEX() 就抽取了前两个元素。然后，外部的 SUBSTRING_INDEX() 再从该结果中抽取最后一个元素。

如果你已经知道了起始位置和抽取长度，那么 SUBSTRING() 会比较好用。如果要在刚才"竖线分隔"的例子中使用 SUBSTRING()，我们得先知道竖线的位置。而要得知竖线的位置，我们还需借助其他搜索函数。下一节将对它们进行介绍。

10.3 搜索字符串及使用长度函数

MySQL 和 MariaDB 在模式搜索方面并不完善。虽然 REGEXP 运算符可做一些模式匹配，但与 PHP 或 Perl 之类的编程语言相比，这还是太弱了。不过，我们还有其他一些能辅助搜索的函数。本节就来看看。

10.3.1 在字符串中找出某段子串的位置

MySQL 和 MariaDB 提供了一些内置函数，可让你在字符串中找出某段子串的位置。

LOCATE() 能返回一个数字，代表子串第一次在字符串中出现时的位置。顺便说一下，在找到一次出现之后，它就会停止往后查找。现在来看一个例子，假设我们想获取 Avocet 鸟（属于 Recurvirostridae 科的岸鸟）的一个列表：

```
SELECT common_name AS 'Avocet'
FROM birds
JOIN bird_families USING(family_id)
WHERE bird_families.scientific_name = 'Recurvirostridae'
AND birds.common_name LIKE '%Avocet%';

+-------------------+
| Avocet            |
+-------------------+
| Pied Avocet       |
| Red-necked Avocet |
| Andean Avocet     |
| American Avocet   |
+-------------------+
```

再假设我们想去掉结果中的 Avocet 字眼。有几种做法：其中一种就是使用 LOCATE() 找出 Avocet 的位置，然后用 SUBSTRING() 抽取该位置之前的文本：

```
SELECT
SUBSTRING(common_name, 1, LOCATE(' Avocet', common_name) ) AS 'Avocet'
FROM birds
JOIN bird_families USING(family_id)
```

```
WHERE bird_families.scientific_name = 'Recurvirostridae'
AND birds.common_name LIKE '%Avocet%';

+-------------+
| Avocet      |
+-------------+
| Pied        |
| Red-necked  |
| Andean      |
| American    |
+-------------+
```

这个例子虽然有点转弯抹角，但我们可以学习到怎样将 LOCATE() 与其他函数配合使用，返回我们想要的结果。下面再看另一个例子。

在 10.1.3 节中，我们试过合并其他观鸟俱乐部的数据。其中涉及了使用 TRIM() 来去掉鸟种学名两端的双引号。现在，再拿它来作例子，不过这次假设它没有双引号了，而是变成了这样的格式：每种鸟的名字都带有其科名，并且鸟种名和科名以两端带有空格的横线（-）隔开。于是，我们就用 LOCATE() 来找出横线的位置，然后用 SUBSTRING() 获取科名，与我们的 birds 表进行连接。以下是 JOIN 部分：

```
…
JOIN rookery.birds
ON(scientific_name = SUBSTRING(science_name, LOCATE(' - ', science_name) + 3 ) );
```

现在来解析这个例子。首先看看 LOCATE()。它所做的是在 science_name 中查找两端带有空格的横线，然后返回该串出现的位置。之后，我们还在其返回结果的基础上加 3，因为该串的长度就是 3 个字符。换句话说，LOCATE() 返回的是该串第一个字符在 science_name 中的位置，而我们要的是紧接着该串的那个字符的位置。因此，SUBSTRING() 的起始位置就是 LOCATE() + 3。而又因为我们没在 SUBSTRING() 中指定抽取长度，所以从 LOCATE() + 3 直至结尾的字符（鸟种的学名）都被抽取出来了。

POSITION() 与 LOCATE() 类似，但它不用逗号来区分子串和整串，而是用 IN：

```
POSITION(' - ' IN science_name)
```

另外，LOCATE() 还有一个可选参数，能让你指定从哪个位置开始搜索，而 POSITION() 则没有这个功能。

FIND_IN_SET() 是另一个搜索函数。如果你有一个内部用逗号分段的字符串，就可以用 FIND_IN_SET() 来查出哪一段含有你所给的模式。为了更好地理解这一点，假设我们有一个按注册日期排序的俄罗斯用户列表。输入以下命令：

```
SELECT human_id,
CONCAT(name_first, SPACE(1), name_last) AS Name,
join_date
FROM humans
WHERE country_id = 'ru'
ORDER BY join_date;
```

```
+----------+------------------+-----------+
| human_id | Name             | join_date |
+----------+------------------+-----------+
|       19 | Elena Bokova     | 2011-05-21 |
|       27 | Anahit Vanetsyan | 2011-10-01 |
|       26 | Katerina Smirnova | 2012-02-01 |
+----------+------------------+-----------+
```

从这个列表可以很容易就看出，Anahit Vanetsyan 是第二位注册的俄罗斯用户。那是因为这个列表很小。但如果列表包含数百个用户呢？我们可以试试改成子查询并配合 FIND_IN_SET() 来查找：

```
SELECT FIND_IN_SET('Anahit Vanetsyan', Names) AS Position
  FROM
    (SELECT GROUP_CONCAT(Name ORDER BY join_date) AS Names
     FROM
       ( SELECT CONCAT(name_first, SPACE(1), name_last) AS Name,
         join_date
         FROM humans
         WHERE country_id = 'ru')
       AS derived_1 )
    AS derived_2;

+----------+
| Position |
+----------+
|        2 |
+----------+
```

这条 SQL 语句比较复杂。最里层的 SELECT 来自之前那个，但这次只返回全名和注册日期。然后，我们将该 SELECT 的结果送给 GROUP_CONCAT，这会产生一个含有所有名字且很长的字符串。最后，最外层的 SELECT 会在这个字符串中查找 Anahit Vanetsyan 的位置。

> 将子查询当成提取表来用时，必须用 AS 给它起一个别名。为了命名简单明了，这里用了 derived_1 和 derived_2。一般来说，只要名字不重复就行。

在用户档案的页面显示信息时，你可能需要这种 SQL 语句。例如，显示用户在某些排行榜（如发现次数最多）中的位置。

如果找不到子串，或者原串为空，则 FIND_IN_SET() 会返回 0。而如果原串是 NULL，则 FIND_IN_SET() 会返回 NULL。

10.3.2 字符串长度

有时你可能会想知道一个字符串有多长。MySQL 提供了一些能返回字符串长度的函数。你可以在调整字符串格式，或对字符串进行某些判断时，配合使用它们。事实上，它们也经常被拿来与 LOCATE() 和 SUBSTRING() 一起使用。

CHAR_LENGTH() 和 CHARACTER_LENGTH() 可返回一个字符串中所含的字符数量，能用于查出多行中某一列字符串的长度。

例如，假设我们准备在 Rookery 网站上展示一些记录在 bird_sightings 表中的最近发现的鸟。显示内容包括俗名、学名等鸟种信息。另外，我们还打算展示用户发现鸟时所写的评论性文字。不过，因为有些评论可能很长，所以我们想先检查每条评论的长度。如果遇到长评论（超过 100 个字符），就截短它，并给出一个超链接让人查看完整内容。想在程序中检查字符串长度，可以这样写：

```
SELECT IF(CHAR_LENGTH(comments) > 100), 'long', 'short')
FROM bird_sightings
WHERE sighting_id = 2;
```

此处用了 CHAR_LENGTH() 来检查所选行的 comments 列的长度，并用 IF() 来判断该长度是否超过 100 个字符。如果是，则返回 long，否则返回 short。如果要将此 SQL 语句用在 API 脚本中，可动态替换 WHERE 中的 sighting_id，以得知每次发现的评论的长度。

CHAR_LENGTH() 判断长度的规则，符合当前字符集（4.1 节讲过字符集的问题）。那些需要多个字节保存的字符（通常来自亚洲语言）也会被当成一个字符来统计。相反，LENGTH() 则是统计字节。注意，一个字节由八位组成，而西方语言通常是一个字母占一个字节。如果想统计位数，可用 BIT_LENGTH()。

比如说，我们发现 bird_sightings 中有些 comments 包含了一些奇怪的二进制字符。它们可能是会员使用移动应用输入的。为了找出哪些行含有这样的字符，以便删掉它们，可以用以下 SQL 语句来检测：

```
SELECT sighting_id
FROM bird_sightings
WHERE CHARACTER_LENGTH(comments) != LENGTH(comments);
```

此 SQL 语句会返回 comments 中字符数与字节数不一致的 sighting_id。

10.3.3　比较和查找字符串

上一小节试过将 CHAR_LENGTH() 的输出作为 IF() 的输入，以决定返回什么值。而本小节将介绍一些比较字符串的函数，它们也很适合用在 IF() 和 WHERE 之中。

先设想一个需要这些函数的场景——准确地说，是需要使用 STRCMP() 的场景。很多程序员都喜欢该函数的名字，它是 string compare（字符串对比）的缩写。

电子邮件是我们与会员联系的重要方式，所以，我们决定要求新加入的会员在注册时输入两次电子邮件地址，以确保填写正确。为了防止在这个过程中网络连接中断，或者新加入的会员没有修正地址中的错误，我们需要有个表来保留这两个地址，直至他们修正。注册时，无论他们输入得是否正确，都先记入 humans 表，然后在另一个表里将两个电子邮件地址记录下来，以做后续对比。而对比的做法，就是用 STRCMP()。

这种对比操作一般会用其他 API（用于与 MySQL/MariaDB 交互的程序）来执行。我们会在该程序中写好用于检查邮件地址的 SQL 语句。不过，一开始我们得先创建一个保存

human_id 和两个电子邮件地址的表，如下：

```
CREATE TABLE possible_duplicate_email
(human_id INT,
email_address1 VARCHAR(255),
email_address2 VARCHAR(255),
entry_date datetime );
```

这样，当新用户的信息被保存到 humans 中之后，我们的程序就会选择性地保存该用户的两个电子邮件地址到 possible_duplicate_email 中。该选择性程序可能如下：

```
INSERT IGNORE INTO possible_duplicate_email
(human_id, email_address_1, email_address_2, entry_date)
VALUES(LAST_INSERT_ID(), 'bobyfischer@mymail.com', 'bobbyfischer@mymail.com')
WHERE ABS( STRCMP('bobbyrobin@mymail.com', 'bobyrobin@mymail.com') ) = 1 ;
```

在上面的语句中，我直接写出了邮件地址。不过在现实环境中，这种 SQL 语句可能是嵌在 PHP 脚本里的，而邮件地址会用变量给出（例如 $email_1 和 $email_2）。

WHERE 中的 STRCMP() 会在两个邮件地址一致时返回 0。如果不一致，则会返回 1 或 −1。−1 是指第一个值的字母序先于第二个值。为了应付这种情况，我们加了 ABS() 函数，以获取 STRCMP() 结果的绝对值。于是，当两个邮件地址不一致时，它们就会被插入 possible_duplicate_email，以便管理员翻查。顺便说一下，该语句会报错，但 IGNORE 会令 MySQL 忽略报错信息。

另一个比较函数是 MATCH() AGAINST()，可用于查找出匹配的行。它甚至可以根据相关性来对每一行进行排名，但这已经超出本章的范围了。它只能在有 FULLTEXT 索引的列上使用。为了试用该函数，我们先在 bird_sightings 表的 comments 列上加 FULLTEXT 索引（因为该列是 TEXT 类型）：

```
CREATE FULLTEXT INDEX comment_index
ON bird_sightings (comments);
```

这样就可以用 MATCH() AGAINST() 了。通常我们会把它作为条件写在 WHERE 中，以找出含有给定字符串的列。给定字符串的内容会根据空格和引号分词。其中，小词（少于或等于三个字符的词）一般会被忽略。以下是例子：

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS Name,
common_name AS Bird,
SUBSTRING(comments, 1, 25) AS Comments
FROM birdwatchers.bird_sightings
JOIN birdwatchers.humans USING(human_id)
JOIN rookery.birds USING(bird_id)
WHERE MATCH (comments) AGAINST ('beautiful');

+-------------------+-----------------+---------------------------+
| Name              | Bird            | Comments                  |
+-------------------+-----------------+---------------------------+
| Elena Bokova      | Eskimo Curlew   | It was a major effort get |
| Katerina Smirnova | Eurasian Curlew | Such a beautiful bird. I  |
+-------------------+-----------------+---------------------------+
```

在此 WHERE 中，我们可以用字符串 beautiful 与 comments 列进行配对。对于匹配的，返回该列以及另外三列：rookery.birds 的 common_name，以及 birdwatchers.humans 的 name_first 和 name_last。

我们还用了 SUBSTRING() 来限制返回内容的长度（即截短内容）。你可以再用 CONCAT() 拼接一些括号，以暗示还有一些文字没显示。还可以用 IF() 来判断是否还有更多内容，以决定是否加括号。如需查出 beautiful 在原文中的位置，并只显示该位置前后的文本，MySQL 也有其他函数可以做到。这个我们会在本章后面讲到。

10.3.4 在字符串中替换或插入内容

如果想在字符串中插入或替换一些内容（并不是替换全部），可以使用 INSERT() 函数。别把它和 INSERT 语句搞混了。它的语法如下：首先指明待插入内容的字符串或列，然后指明插入的位置。也可以指定想删除多少内容。最后是要插入的内容。下面来看例子。

先看一个简单的例子。假设我们想在 Rookery 网站的页面上，给鸟种俗名中的 Least 加上注释 Smallest，以让那些不太懂行的观鸟者知道 Least 的意思是“最小”，而非“最次要”。做法如下：

```
SELECT INSERT(common_name, 6, 0, ' (i.e., Smallest)')
AS 'Smallest Birds'
FROM birds
WHERE common_name LIKE 'Least %' LIMIT 1;

+------------------------------+
| Smallest Birds               |
+------------------------------+
| Least (i.e., Smallest) Grebe |
+------------------------------+
```

第一个参数就是我们要修改的列。第二个就是插入的起始位置。而根据 WHERE 条件，我们查的是俗名以 Least 开头的鸟。因为 Least 是 5 个字符，所以我们还要加 1，使插入的内容放在 Least 后的空格之后。第三个参数是指起始位置之后有多少个字符应被替换。不过我们现在只插入而不替换。

INSERT() 的效果体现在结果集中，它不会更改表中的数据。我们大可用 INSERT() 来改变俗名的格式，让那些自认为是观鸟新手的用户在头一个月熟悉各种鸟名。要想编写查找新手的 SQL 语句，就要再复杂一些了。因为我们这里只讲解 INSERT() 的用法，所以就不写了。现在来看看如何用 INSERT() 替换字符串中的数据。

假设我们发现 birds 表中有些鸟的俗名含有缩写（比如，将 Great 缩写成 Gt. 了）。现在我们打算用 INSERT() 把缩写词扩展回去。在修改数据之前，先用 SELECT 来看看 INSERT() 写得是否正确：

```
SELECT common_name AS Original,
INSERT(common_name, LOCATE('Gt.', common_name), 3, 'Great') AS Adjusted
FROM birds
WHERE common_name REGEXP 'Gt.' LIMIT 1;
```

```
+------------------+--------------------+
| Original         | Adjusted           |
+------------------+--------------------+
| Gt. Reed-Warbler | Great Reed-Warbler |
+------------------+--------------------+
```

我们已在之前的例子中回顾了 INSERT() 各个参数的意义。而本例有新意的地方，就是第二个参数使用了 LOCATE()。它的目的是查出待替换文本所在的位置。之前的例子假设 Least 只会出现在开头。但本例则不这样猜测 Gt. 的位置，相反，我们用函数来查找其位置。

本例的另一个不同之处，就是第三个参数：我们告诉这个函数，把 Gt. 的起始位置往后的三个字符（即 Gt. 的长度），替换成 Great（由第四个参数指定）。尽管用于替换的串长于原串，但该列还是有足够的空间容纳。

如果 LOCATE() 没找到 Gt.，那么就会返回 0。而 0 作为 INSERT() 的第二个参数时，将使 INSERT() 直接返回原内容。所以，我们没必要用 WHERE 来筛选出含 Gt. 的行——除非你仅想显示这些行。

既然已经确认此 INSERT() 没问题，那么就可以开始修改数据了：

```
UPDATE birds
SET common_name = INSERT(common_name, LOCATE('Gt.', common_name), 3, 'Great')
WHERE common_name REGEXP 'Gt.';
```

除了用 INSERT()，我们还有其他方法来替换字符串。使用 INSERT() 时，必须带上 LOCATE()，以决定在哪里进行替换，并且，还得指明替换多少个字符。而使用 REPLACE() 的话，就简单了。我们可以用它来将 common_name 中所有的 Gt. 替换成 Great。下面就用 SELECT 来测试：

```
SELECT common_name AS Original,
REPLACE(common_name, 'Gt.', 'Great') AS Replaced
FROM birds
WHERE common_name REGEXP 'Gt.' LIMIT 1;

+------------------+--------------------+
| Original         | Replaced           |
+------------------+--------------------+
| Gt. Reed-Warbler | Great Reed-Warbler |
+------------------+--------------------+
```

下面这种做法更好。我们可以用带着这些参数的 REPLACE( )，并且输入以下 UPDATE 来实施数据更改：

```
UPDATE birds
SET common_name = REPLACE(common_name, 'Gt.', 'Great');

Query OK, 8 rows affected (0.23 sec)
Rows matched: 28891  Changed: 8  Warnings: 0
```

注意，我们没有带上 WHERE，但结果还是只修改了八条。那是因为只有这八行的 common_name 含有 Gt.。更新含有多行数据的表时，不加 WHERE 是很危险的，所以最好还是先用 SELECT 测测这些参数。

10.4 转换字符串类型

很多时候，我们都会遇到一些他人没有定义好数据类型的表。有时，你可以改表结构，但有时不被允许这么做。面对这样的表，可以使用 CAST() 或 CONVERT() 函数来改变列的数据类型。这两个函数只影响结果集，不会更改原数据。其实它们基本上是一样的，只是语法有些许差异。接下来，我们会拿一些例子来看看如何以及为何要使用它们。

假设我们有一个用于保存鸟种图像信息的表，从那些图像可看出每种鸟雌性、雄性和幼年的颜色。该表有一列是图像的序号，它是根据鸟种和拍摄日期而定的。不过该列没用 INT 型，而用了 CHAR。如果以该列来给数据排序的话，MySQL 就会按词法序来排，而不是按数字序，如下所示：

```
SELECT sorting_id, bird_name, bird_image
FROM bird_images
ORDER BY sorting_id
LIMIT 5;

+------------+----------------+---------------------------+
| sorting_id | bird_name      | bird_image                |
+------------+----------------+---------------------------+
| 11         | Arctic Loon    | artic_loon_male.jpg       |
| 111        | Wilson's Plover | wilson_plover_male.jpg   |
| 112        | Wilson's Plover | wilson_plover_female.jpg |
| 113        | Wilson's Plover | wilson_plover_juvenile.jpg |
| 12         | Pacific Loon   | pacific_loon_male.jpg     |
+------------+----------------+---------------------------+
```

注意，所有 sorting_id 为 11 开头的行，都被排在 12 的前面了。因为 MySQL 将它们当成了字符，而非数字。如果排序正确，那两个 Loon 应该是在一起的，并且排在 Plover 之前。

我们可以用 CAST() 将 sorting_id 的值换成 INT 类型：

```
SELECT sorting_id, bird_name, bird_image
FROM bird_images ORDER BY CAST(sorting_id AS INT) LIMIT 5;

+------------+----------------+---------------------------+
| sorting_id | bird_name      | bird_image                |
+------------+----------------+---------------------------+
| 11         | Arctic Loon    | artic_loon_male.jpg       |
| 12         | Pacific Loon   | pacific_loon_male.jpg     |
| 111        | Wilson's Plover | wilson_plover_male.jpg   |
| 112        | Wilson's Plover | wilson_plover_female.jpg |
| 113        | Wilson's Plover | wilson_plover_juvenile.jpg |
+------------+----------------+---------------------------+
```

现在排序正确了。再假设我们不想按 sorting_id 排了，而想按 gender_age 排。该列是 ENUM 型的，可选 male、female 或 juvenile。鸟的颜色类型主要因这三个因素而异。如果按这列来排序的话，结果如下：

```
SELECT bird_name, gender_age, bird_image
FROM bird_images
WHERE bird_name LIKE '%Plover%'
```

```
ORDER BY gender_age
LIMIT 5;

+-----------------+------------+---------------------------+
| bird_name       | gender_age | bird_image                |
+-----------------+------------+---------------------------+
| Wilson's Plover | male       | wilson_plover_male.jpg    |
| Snowy Plover    | male       | snowy_plover_male.jpg     |
| Wilson's Plover | female     | wilson_plover_female.jpg  |
| Snowy Plover    | female     | snowy_plover_female.jpg   |
| Wilson's Plover | juvenile   | wilson_plover_juvenile.jpg |
+-----------------+------------+---------------------------+
```

可看出它们按 gender_age 归类了，但顺序不是字母序（female 本应在 male 之前），而是按每个枚举值定义的顺序：

```
SHOW COLUMNS FROM bird_images LIKE 'gender_age' \G

*************************** 1. row ***************************
  Field: gender_age
   Type: enum('male','female','juvenile')
   Null: YES
    Key:
Default: NULL
  Extra:
```

这样，对 MySQL 来说，male 就是 1，female 就是 2，所以结果集中，male 排在 female 之前，尽管字母序不是这样。为了修正，可以在 ORDER BY 中用 CAST() 或 CONVERT()，这样 MySQL 就会根据函数的返回值，而不是列的原值来排。具体如下：

```
SELECT bird_name, gender_age, bird_image
FROM bird_images
WHERE bird_name LIKE '%Plover%'
ORDER BY CONVERT(gender_age, CHAR)
LIMIT 5;

+-----------------+------------+---------------------------+
| bird_name       | gender_age | bird_image                |
+-----------------+------------+---------------------------+
| Wilson's Plover | female     | wilson_plover_female.jpg  |
| Snowy Plover    | female     | snowy_plover_female.jpg   |
| Wilson's Plover | juvenile   | wilson_plover_juvenile.jpg |
| Snowy Plover    | juvenile   | snowy_plover_juvenile.jpg  |
| Wilson's Plover | male       | wilson_plover_male.jpg    |
+-----------------+------------+---------------------------+
```

注意，在 CONVERT() 中，我们不是用 AS 来分隔原值和新类型，而是用逗号。第二个参数的新类型可以是 BINARY、CHAR、DATE、DATETIME、SIGNED [INTEGER]、TIME 或 UNSIGNED [INTEGER]。其中，BINARY 会将字符串转换成二进制串。你也可以加上 CHARACTER SET，以进行字符集转换。要转换给定字符串的字符集，需要使用 USING 选项，如下：

```
SELECT bird_name, gender_age, bird_image
FROM bird_images
```

```
WHERE bird_name LIKE '%Plover%'
ORDER BY CONVERT(gender_age USING utf8)
LIMIT 5;
```

10.5 压缩字符串

有些列类型可包含大量数据，例如 BLOB。为了减少使用了这种类型的表的占用空间，可以在插入数据时，先对数据进行压缩。压缩用 COMPRESS()，解压缩用 UNCOMPRESS()。要使用这两个函数，必须让 MySQL 安装时带上压缩库（即 zlib）。如果没有的话，使用 COMPRESS() 时就会返回 NULL。现在来看一些例子。

humans 表中有一列叫 birding_background，它是 BLOB 型。会员可在其中填写任何关于其观鸟经验的内容，最终这些信息会在网上显示。如果很多人同时填写的话，那有可能导致查询和更新变慢。所以我们决定，在往 humans 表插入这些信息前，先用 COMPRESS() 压缩。做法如下：

```
INSERT INTO humans
(formal_title, name_first, name_last, join_date, birding_background)
VALUES('Ms', 'Melissa', 'Lee', CURDATE(), COMPRESS("lengthy background..."));
```

此 SQL 语句会把一个新用户的信息录到 humans 表中——该表还有很多列，不过为了简化这个例子，就先写这么多。它用了 COMPRESS() 来将背景信息压缩（虽然本例中的背景信息不是很长）。背景信息可以来自网页的用户输入，然后保存在其他 API 程序（如 PHP）的变量中（如 $birding_background），再传给 SQL 语句。

想看压缩后的样子，可这样查询：

```
SELECT birding_background AS Background
FROM humans
WHERE name_first = 'Melissa' AND name_last = 'Lee' \G

*************************** 1. row ***************************
Background:    x#####/ŧTHJL##/######## Z######
```

注意，它看起来不像平常的文本。那是因为 mysql 将二进制值换成了 #。要想看原本的内容，可用 UNCOMPRESS()。如果存储的值是没压缩过的，或 MySQL 没有带上 zlib 一起编译，则它会返回 NULL：

```
SELECT UNCOMPRESS(birding_background) AS Background
FROM humans
WHERE name_first = 'Melissa' AND name_last = 'Lee' \G

*************************** 1. row ***************************
Background: lengthy background...
```

对于这种少量的文本，压缩后比压缩前更费空间。但对于大量文本，则会节省空间。所以，请谨慎使用压缩函数。

10.6　小结

MySQL 和 MariaDB 其实还有更多可用的字符串函数。其中有些是这里提到的一些函数的别名或功能类似的替代品。有些是能将字符串在 ASCII、二进制、十六进制和八进制之间转换的函数。还有些是我们未提及的加密和解密函数。不过，我认为本章已经将最常用的都介绍完了，它们足以助你构建出更强大的 SQL 语句，生成更炫的样式。

10.7　习题

对于开发 MySQL 和 MariaDB 的数据库来说，字符串函数非常重要。你应该很好地掌握它们。要想成为这类函数的专家，得多加练习。所以，请确保完成以下习题。

(1) `CONCAT()` 是最常用的字符串函数之一。构建一条查询 `humans` 表的 `SELECT` 语句。用 `CONCAT()` 将 `name_first` 与 `name_last` 拼接在一起，并用 `SPACE()` 在两者之间加空格。为结果取别名 `Full Name`——记得定义这个别名时要加引号，因为它包含空格。只查四行，并确保运行正常。

给 `SELECT` 加一个 `WHERE` 子句，并在其中复制刚才所写的 `CONCAT()`，以限定只查出这些人：Lexi Hollar、Michael Zabalaoui 和 Rusty Johnson。

上面运行没问题的话，再加一个 `ORDER BY`，使结果集按拼接后的名字排序。不能用到 `CONCAT()`。

(2) 构建一条 `SELECT` 语句，查询 `birds` 表的 `common_name` 和 `scientific_name`。用字符串函数将 `scientific_name` 改成全部小写。再使用 `CONCAT()` 将它们拼接成一个域，格式是俗名后留一个空格，接着是加括号的学名，如 African Desert Warbler (sylvia deserti)。空格不要用 `SPACE()`。空格和括号都用单引号包围，拼进 `CONCAT()` 中。拼接后，起别名为 `Bird Species`。将结果限制为 10 行。

执行成功后，将这条 SQL 语句改成与 `bird_families` 和 `bird_orders` 连接。可到 9.2 节中参考 `JOIN` 的用法。最终结果要包含这些表的 `scientific_name`。

确认可以运行后，将新加的两个表的 `scientific_name` 移到 `CONCAT()` 中。使用 `RPAD()` 在种名后，科名和目名前，添加一些句点。类似这样：

```
Speckled Warbler (pyrrholaemus sagittatus)...Acanthizidae...Passeriformes
```

你可能会需要使用两次 `CONCAT()`。最后，再用 `WHERE` 选出 `Warbler` 的那些行，并只显示 10 行。

(3) 再构建一条 `SELECT` 语句，查出所有俗名含 Shrike 的鸟种。你会在结果中看到有些 Shrike 带有横杠。用 `REPLACE()` 将横杠替换成空格，然后再执行一次。

(4) 刚才那个习题还会查出一些名字带有两个横杠的鸟（例如 Yellow-browed Shrike-Vireo）。重做该习题，使只有 Shrike 后的横杠被替换（例如 Yellow-browed Shrike Vireo）。要做到这种效果，用 `LOCATE()` 配合 `REPLACE()`。`LOCATE()` 要写两次，一个 `LOCATE` 包含另一个。

(5) Laniidae 科的 Shrike，才是真的 Shrike。构建一条 SELECT 语句，搜索俗名含有 Shrike 并且属于 Laniidae 的鸟。你会需要连接 bird_families 表。然后，使用某个抽取字符串的函数，如 SUBSTRING()，将 Shrike 前的词抽取出来。这需要 LOCATE() 之类的函数配合。接着，将截出的词放到 Shrike 后面，用逗号和空格隔开。结果应该会变成这样：Shrike, Rufous-tailed。给该域起别名为 Shrikes。

(6) 在 humans 表里，会员输入的姓名有大写的，也有小写的（例如 andy oram 和 MICHAEL STONE）。用 UPDATE 将它们改成标题样式（即首字母大写，其余小写）。先用 SELECT 测试函数组合得是否正确。更改大小写，你会用到 UCASE() 和 LCASE()。还会多次使用 SUBSTRING() 之类的函数，偶尔还会用到 CONCAT()。

第 11 章

日期和时间函数

大多数人都将每天分为上午和下午，而将两个 12 小时或一个 24 小时当成一天。一年有 12 个月，除了其中一个月份只有 28 天（不过每四年它会变成 29 天），其他月份都有 30 天或 31 天。这对于人类来说是很自然的事，或者至少是很熟悉的事。但对于计算机来说，却不是这样。然而，在数据库中存储并操作日期和时间又是一个很常见的需求。

要在数据库中存储日期和时间（即时间型数据），你得了解在表里有什么列类型可用。更重要的是，要了解如何把时序数据存进数据库，以及如何以其他格式获取数据。虽然这听起来很简单，但 MySQL 和 MariaDB 提供了很多内置时间函数，能让你把 SQL 语句构建得更准确，把数据格式化得更美观。在本章中，我们来探讨这些函数。

11.1 日期和时间的数据类型

因为日期和时间不过是包含数字的字符串，所以可用普通的字符型来存储。然而，我们有专为日期和时间而设计的数据类型。如果将日期和时间保存为这样的类型，我们就可以利用一些内置函数来操作它们。所以，在学习这些函数之前，让我们先看看有些什么日期和时间类型。

MySQL 和 MariaDB 有五种时间型数据：存储日期的 `DATE`，存储时间的 `TIME`，将日期和时间一起存储的 `DATETIME` 和 `TIMESTAMP`，以及存储年份的 `YEAR`。

- `DATE`

  它只保存日期，格式为 yyyy-mm-dd。你可能更喜欢其他格式（例如，将情人节显示成 02-14-2014），但 `DATE` 的存储方式不可改变——至少，无法在不改变 MySQL 源代码的情况下，改变其存储方式。如果想按其他格式来显示日期，可以用本章介绍的其他函数。

DATE 对其可接受的日期范围有限制，即 1000-01-01 至 9999-12-31。这足以保存很久以后的日期，虽然它不能用来保存第一个千年里的日期。

- TIME

  它能将时间按 hhh:mm:ss 的格式保存。可接受的时间范围是 -838:59:59 至 838:59:59。如果你给的数据超出了该范围，或者给的是无效数据，那么它会全部被保存成 0。
  你可能会想不通为何需要三位数来表示小时。其实那是为了方便表示一件事情的持续时间，或对比两个时间，而不只是保存一天中的某个时间。例如，当你想记录“某件事花了 120 个小时”时，可以用两列，一列记录开始时间，另一列记录结束时间，然后两者相减。实际上 TIME 可以让你直接保存这个结果，而不需在每次查询时重新计算。

- DATETIME

  它能将日期和时间保存在一起，格式为 yyyy-mm-dd hh:mm:ss。可接受的日期和时间范围是 1000-01-01 00:00:00 至 9999-12-31 23:59:59。其中日期的范围与 DATE 一样，而时间则是 24 小时。在 MySQL 5.6 版本中，甚至可以比秒更精确。

- TIMESTAMP

  它与 DATETIME 类似，但范围更有限。虽然名字中有 TIME，但却不是说在一天的某个时间段上有限制，而是在日期上，它只能保存 1970-01-0100:00:01 UTC 至 2038-01-19 03:14:07 UTC 之间的日期和时间。这适用于从 Unix 时间戳到我们现在这个时代的时间。而在 MySQL 5.6 版本中，甚至可以比秒更精确。
  当列被定为这种类型时，除了手动给其录入或更新日期和时间，也可以让 MySQL 自动为你录入或更新为当前日期和时间。这对于日志程序之类的应用来说，十分方便。但注意，一般能自动录入的只有表中的第一个 TIMESTAMP 列，如果想让其他 TIMESTAMP 列也获得同样效果，需要设定 ON UPDATE CURRENT_TIMESTAMP 和 ON INSERT CURRENT_TIMESTAMP。

- YEAR

  它只保存年份，格式为 yyyy。你也可以将它设为只保存两位数（YEAR(2)）。不过这会导致一些问题，所以不推荐这么做。
  它也可以表示人的出生年份，可接受的范围是 1901 至 2155。如果给的值无效或超出该范围，则会被保存成 0000。

> 因为有以上这些限制，所以当需要存储超出范围的日期时，你可能要用到非时间型的列类型。例如，可将年月日拆成三个 INT 列来存储，或者合在一起，保存成固定格式的 CHAR，又或者是月和日用 INT，年用 CHAR(4)（这样就可以解决 YEAR 不能保存 20 世纪之前年份的问题）。
>
> 一般来说这样是没问题的，但如果要进行日期计算，就麻烦了。假设你用两个 INT 列保存 2 月 15 日：2 放在 my_month，15 放在 my_day。然后，你想给该日期加 20 天。但直接加的话，就会得出 2 月 35 日这样的无效日期。如此一来，你得写一个很复杂的日期判断逻辑（还要考虑年份的变动）。同样，用 INT 保存时间也会遇到这种状况。而用时间型，再配合 MySQL 的日期和时间函数，则无需担心这个问题，因为这些函数已封装了那些复杂的计算。

熟悉过这些类型并知道了它们的优点之后，就可以看一些例子，学习怎样用日期和时间函数来操作它们。因为我们之前建好的表已定义了上述数据类型，所以就直接拿它们来用吧。

11.2　当前日期和时间

最基本的日期和时间函数，就是那些与当前日期和时间相关的函数。你可以用它们来保存当前日期和时间，或修改基于当前日期和时间的结果，或在结果集中显示日期和时间。现在先试试最简单的一个函数，NOW()。你在哪个时间点执行它，它就返回哪个时间点。输入下例的第一行（后面的是结果）：

```
SELECT NOW( );

+---------------------+
| NOW( )              |
+---------------------+
| 2014-02-08 09:43:09 |
+---------------------+
```

如你所见，它返回了服务器上的日期和时间，格式和 DATETIME 的格式一致。所以，如果某个表中有 DATETIME 列，那么就可以很方便地用 NOW() 给该列录入当前日期和时间。例如，**bird_sightings** 表的 **time_seen** 列就是这样。以下是给该表插入一行数据时，使用 NOW() 的例子：

```
INSERT INTO bird_sightings
(bird_id, human_id, time_seen, location_gps)
VALUES (104, 34, NOW( ), '47.318875; 8.580119');
```

你也可以在应用程序或服务器脚本中使用该函数，这样用户在记录发现鸟的情况时，就可以不需要自己录入时间信息了。

> NOW() 有几个同义词：CURRENT_TIMESTAMP()、LOCALTIME() 和 LOCALTIMESTAMP()。它们返回的结果都是一样的。MySQL 和 MariaDB 提供这些同义词，是为了兼容运行于其他 SQL 数据库的函数。如果你原本用的是 PostgreSQL、Sybase 或 Oracle，都可以很轻松地将它替换成 MySQL，而无需修改 SQL 语句。

NOW() 所返回的日期和时间是它所在的 SQL 语句开始执行时的日期和时间。大多数情况用它都没问题：如果 SQL 语句开始和结束的时间点极其相近，又或者你并不介意取开始时间或结束时间。但如果该 SQL 语句运行时间特别长，而你想记录这个过程中的某个时间点，可使用 SYSDATE()，它返回的是自身被执行时的那个时间点（不是整条语句的结束时间）。想看看它的特别之处，可引入 SLEEP()，让 MySQL 在 SQL 语句的执行过程中暂停指定的秒数。下面用一个简单的例子来展示 NOW() 和 SYSDATE() 的区别：

```
SELECT NOW(), SLEEP(4) AS 'Zzz', SYSDATE(), SLEEP(2) AS 'Zzz', SYSDATE();

+---------------------+-----+---------------------+-----+---------------------+
| NOW()               | Zzz | SYSDATE()           | Zzz | SYSDATE()           |
```

```
+---------------------+-----+---------------------+-----+---------------------+
| 2014-02-21 05:44:57 |   0 | 2014-02-21 05:45:01 |   0 | 2014-02-21 05:45:03 |
+---------------------+-----+---------------------+-----+---------------------+

1 row in set (6.14 sec)
```

注意，NOW() 的结果和第一个 SYSDATE() 的结果相差 4 秒，这也是第一个 SLEEP() 所指定的秒数。而 SLEEP(2) 则使两个 SYSDATE() 的结果差了 2 秒。另外，我们也留意到，结果集之后的提示信息表示，执行整条 SQL 语句所花的时间多于 6 秒。你应该很少会用到 SYSDATE()——甚至完全不用。它主要用在一些非常复杂的 SQL 语句中，或者是存储过程和触发器之中。下面，我们还是回到一般的用途上。

即使列不是 DATETIME 类型，也并不妨碍我们将 NOW() 的返回值保存到列中。比如说，如果 time_seen 是 DATE 类型，那么，若再执行之前那条 INSERT 语句，它就会报告说数据被截断。不过，日期部分还是能正确地保存进去。同理，如果列是 TIME 类型，那么也会出现数据被截断的警告，但时间部分还是正确地保存下来。尽管如此，我们最好还是使用匹配的函数：用 CURDATE() 的值填充 DATE 列，而用 CURTIME() 的值填充 TIME 列。以下例子对比三个函数的差异：

```
SELECT NOW( ), CURDATE( ), CURTIME( );

+---------------------+------------+-------------+
| NOW( )              | CURDATE( )  | CURTIME( ) |
+---------------------+------------+-------------+
| 2014-02-08 10:23:32 | 2014-02-08 | 10:23:32    |
+---------------------+------------+-------------+
```

这三个函数以及它们的同义词所返回的格式都很明了。不过，也有内置函数返回的是 Unix 时间，这种格式显示的是从 Unix 时间戳（前面说过）至今的秒数。当需要比较两个时间型时，这种格式很有用。以下是 NOW() 的 TIMESTAMP 形式：

```
SELECT UNIX_TIMESTAMP( ), NOW( );

+-------------------+---------------------+
| UNIX_TIMESTAMP( ) | NOW( )              |
+-------------------+---------------------+
|        1391874612 | 2014-02-08 10:50:12 |
+-------------------+---------------------+
```

这里的秒数从 1970 年 1 月 1 日起计。现在来测试一下，看它到底对不对。我们先用一种简单的计算方法，得出 1970 年至今过了多少年，再用比较复杂的 UNIX_TIMESTAMP() 来算：

```
SELECT (2014 - 1970) AS 'Simple',
UNIX_TIMESTAMP( ) AS 'Seconds since Epoch',
ROUND(UNIX_TIMESTAMP( ) / 60 / 60 / 24 / 365.25) AS 'Complicated';

+--------+---------------------+-------------+
| Simple | Seconds since Epoch | Complicated |
+--------+---------------------+-------------+
|     44 |          1391875289 |          44 |
+--------+---------------------+-------------+
```

因为这是在 2014 年初执行的，所以我们用了 ROUND() 来舍去多出的几个月所产生的小数部分。这样做测试是有意义的，你应该能够更加明白这些函数的用途，也更相信它们的准确性。

我们再看一个可能需要用到 Unix 时间的场景。假设你想知道观鸟者在多少天之前发现了 Black Guineafowl（bird_id 为 309），可以通过连接 bird_sightings 和 humans 这两个表来得到答案：

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
ROUND((UNIX_TIMESTAMP( ) - UNIX_TIMESTAMP(time_seen)) / 60 / 60 / 24)
   AS 'Days Since Spotted'
FROM bird_sightings JOIN humans USING(human_id)
WHERE bird_id = 309;

+-------------+--------------------+
| Birdwatcher | Days Since Spotted |
+-------------+--------------------+
| Marie Dyer  |                129 |
+-------------+--------------------+
```

此例先用 CONCAT() 拼接出了观鸟者的姓名，然后用第一个不带参数的 UNIX_TIMESTAMP() 获取当前日期和时间，接着再用一个 UNIX_TIMESTAMP() 来指定 time_seen 列（这列包含观鸟者发现每种鸟的日期）。这个函数将值改成了 Unix 时间格式，方便我们进行相减。

比较日期和时间的方法和函数还有很多。我们会在本章后面看到。接下来我们先看看如何抽取日期和时间中的某部分。

11.3　抽取日期和时间中的某部分

有时候，时间型数据所保存的信息可能多于你想要的。例如，你不想要完整的日期或时间。为应付这种情况，MySQL 提供了一些用于抽取日期和时间中某部分的函数，以及一些用于指定抽取哪部分、按什么格式返回的代码。现在先看一些仅抽取日期或仅抽取时间的基本函数，之后再看抽取得更细的那些。

DATETIME 类型的列，顾名思义，同时包含了日期和时间。如果只想从中抽取日期，可以用 DATE() 函数。如果只想抽取时间，则可以用 TIME()。下面看看它们的用法。我们还是以 Black Guineafowl 的发现时间来举例：

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
time_seen, DATE(time_seen), TIME(time_seen)
FROM bird_sightings
JOIN humans USING(human_id)
WHERE bird_id = 309;

+-------------+---------------------+-----------------+-----------------+
| Birdwatcher | time_seen           | DATE(time_seen) | TIME(time_seen) |
+-------------+---------------------+-----------------+-----------------+
| Marie Dyer  | 2013-10-02 07:39:44 | 2013-10-02      | 07:39:44        |
+-------------+---------------------+-----------------+-----------------+
```

很简单，DATE() 只返回了 time_seen 的日期部分，而 TIME() 则只返回了时间部分。如果想抽取日期或时间中的某一元素，那也没问题，只要它真的含有该元素——你总不能从 YEAR 中抽取小时吧。

想抽取小时，可以用 HOUR()。而对于分和秒，则有 MINUTE() 和 SECOND()。它们都接受 DATETIME、TIME 和 TIMESTAMP 作为参数。来看看它们返回的结果会是怎样。在 mysql 中输入以下命令：

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
time_seen, HOUR(time_seen), MINUTE(time_seen), SECOND(time_seen)
FROM bird_sightings JOIN humans USING(human_id)
WHERE bird_id = 309 \G

*************************** 1. row ***************************
      Birdwatcher: Marie Dyer
        time_seen: 2013-10-02 07:39:44
  HOUR(time_seen): 7
MINUTE(time_seen): 39
SECOND(time_seen): 44
```

有了这些函数，你就可以使用、估算和比较时间中的各个元素。同理，日期也是可以拆分的。

想从日期中抽出年月日，可以分别使用 YEAR()、MONTH() 和 DAY()。它们接受日期值或含日期的字符串值（例如 '2014-02-14'，注意带有引号）作为参数。而数字的情况则有固定的格式要求。例如 20140214 是可以的，而 2014-02-14（没有引号）就不可以，2014 02 14（有空格）也不可以。下面把刚才的 SQL 语句改改，换上这些函数：

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
time_seen, YEAR(time_seen), MONTH(time_seen), DAY(time_seen),
MONTHNAME(time_seen), DAYNAME(time_seen)
FROM bird_sightings JOIN humans USING(human_id)
WHERE bird_id = 309 \G

*************************** 1. row ***************************
         Birdwatcher: Marie Dyer
           time_seen: 2013-10-02 07:39:44
     YEAR(time_seen): 2013
    MONTH(time_seen): 10
      DAY(time_seen): 2
MONTHNAME(time_seen): October
  DAYNAME(time_seen): Wednesday
```

这里还用到了 MONTHNAME() 和 DAYNAME()，它们能分别告诉你指定日期是一年中的哪个月和一周中的哪一天。使用这些函数，你就能将结果集整理得更美观，让人看得更方便。下面就试试用它们来重新组织日期信息。以下例子将查出会员发现的濒危鸟种：

```
SELECT common_name AS 'Endangered Bird',
CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
CONCAT(DAYNAME(time_seen), ', ', MONTHNAME(time_seen), SPACE(1),
  DAY(time_seen), ', ', YEAR(time_seen)) AS 'Date Spotted',
CONCAT(HOUR(time_seen), ':', MINUTE(time_seen),
```

```
  IF(HOUR(time_seen) < 12, ' a.m.', ' p.m.')) AS 'Time Spotted'
FROM bird_sightings
JOIN humans USING(human_id)
JOIN rookery.birds USING(bird_id)
JOIN rookery.conservation_status USING(conservation_status_id)
WHERE conservation_category = 'Threatened' LIMIT 3;

+--------------------+-------------+--------------------------+-----------+
| Endangered Bird    | Birdwatcher | Date Spotted             | Time      |
+--------------------+-------------+--------------------------+-----------+
| Eskimo Curlew      | Elena Bokova | Tuesday, October 1, 2013   | 5:9 a.m.  |
| Red-billed Curassow | Marie Dyer   | Wednesday, October 2, 2013 | 7:39 a.m. |
| Red-billed Curassow | Elena Bokova | Wednesday, October 2, 2013 | 8:41 a.m. |
+--------------------+-------------+--------------------------+-----------+
```

这条 SQL 语句比较复杂。不过因为用了好几次 JOIN，所以确实需要这么长（但看着那一大串 CONCAT() 以及日期和时间函数，你可能会想，应该有更简单的做法吧）。注意，其中时和分的结果是 5:9，而不是 5:09，这是因为 MINUTE() 不会在左边添加 0。你当然可以用 LPAD() 来修改，但这会搞得更复杂。而 IF() 则用来标记时间是上午或下午（即 a.m. 或 p.m.）。

我们有更简洁的方法来格式化日期和时间，那就是使用格式化函数。下一节会介绍它们。现在，你可以用 EXTRACT() 来替代上述抽取函数。

EXTRACT() 可用于抽取日期或时间中的任何部分。它的语法很简单，但有点冗长：EXTRACT(interval FROM date_time)。其中，interval 与我们之前看过的那些日期和时间抽取函数的名字相似：MONTH 是月份，HOUR 是小时，以此类推。此外，还有一些组合起来的函数，比如 YEAR_MONTH 和 HOUR_MINUTE。想知道 EXTRACT() 的各种 interval 以及类似的函数，见表 11-1。

表11-1：Interval及返回格式

Interval	返回格式	Interval	返回格式
DAY	dd	MINUTE	mm
DAY_HOUR	'dd hh'	MINUTE_MICROSECOND	'mm.nn'
DAY_MICROSECOND	'dd.nn'	MINUTE_SECOND	'mm:ss'
DAY_MINUTE	'dd hh:mm'	MONTH	mm
DAY_SECOND	'dd hh:mm:ss'	QUARTER	qq
HOUR	hh	SECOND	ss
HOUR_MICROSECOND	'hh.nn'	SECOND_MICROSECOND	'ss.nn'
HOUR_MINUTE	'hh:mm'	WEEK	ww
HOUR_SECOND	'hh:mm:ss'	YEAR	yy
MICROSECOND	nn	YEAR_MONTH	'yy-mm'

现在试着用 EXTRACT() 来改写“谁在什么时候发现了 Black Guineafowl”这个例子：

```
SELECT time_seen,
EXTRACT(YEAR_MONTH FROM time_seen) AS 'Year & Month',
EXTRACT(MONTH FROM time_seen) AS 'Month Only',
```

```
EXTRACT(HOUR_MINUTE FROM time_seen) AS 'Hour & Minute',
EXTRACT(HOUR FROM time_seen) AS 'Hour Only'
FROM bird_sightings JOIN humans USING(human_id)
LIMIT 3;

+---------------------+--------------+------------+---------------+-----------+
| time_seen           | Year & Month | Month Only | Hour & Minute | Hour Only |
+---------------------+--------------+------------+---------------+-----------+
| 2013-10-01 04:57:12 |       201310 |         10 |           457 |         4 |
| 2013-10-01 05:09:27 |       201310 |         10 |           509 |         5 |
| 2013-10-01 05:13:25 |       201310 |         10 |           513 |         5 |
+---------------------+--------------+------------+---------------+-----------+
```

如你所见，用 EXTRACT() 单独显示某个部分的话，看起来与之前那些日期和时间抽取函数一样，没什么问题。但 HOUR_MINUTE 却没有在时和分之间加上冒号（4:57 变成了 457），这就不太符合人们的阅读习惯了。用 EXTRACT() 输出元素组合时，它会直接将两个元素拼在一起，不会进行格式化。可能有时你就想要这种效果，但更多时候这是非主流的。所以，你还需要学习下一节的日期和时间格式化函数。

11.4 格式化日期和时间

在 11.1 节中，我简要介绍过 MySQL 和 MariaDB 中的日期和时间数据类型，以及它们的保存格式。我也说过，如果你不喜欢这些格式，可以用一些内置函数来将时间数据转换成其他显示格式。其中，最好用的函数就是 DATE_FORMAT() 和 TIME_FORMAT()。它们可以处理来自列、字符串或其他函数的日期和时间值，你只需指定自己想要的格式的代码即可。现在试试用这两个函数改写上一节最后的那个例子：

```
SELECT common_name AS 'Endangered Bird',
CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
DATE_FORMAT(time_seen, '%W, %M %e, %Y') AS 'Date Spotted',
TIME_FORMAT(time_seen, '%l:%i %p') AS 'Time Spotted'
FROM bird_sightings
JOIN humans USING(human_id)
JOIN rookery.birds USING(bird_id)
JOIN rookery.conservation_status USING(conservation_status_id)
WHERE conservation_category = 'Threatened' LIMIT 3;

+---------------------+--------------+----------------------------+-----------+
| Endangered Bird     | Birdwatcher  | Date Spotted               | Time      |
+---------------------+--------------+----------------------------+-----------+
| Eskimo Curlew       | Elena Bokova | Tuesday, October 1, 2013   | 5:09 AM   |
| Red-billed Curassow | Marie Dyer   | Wednesday, October 2, 2013 | 7:39 AM   |
| Red-billed Curassow | Elena Bokova | Wednesday, October 2, 2013 | 8:41 AM   |
+---------------------+--------------+----------------------------+-----------+
```

整条 SQL 语句依然很冗长，但日期和时间格式化的相关代码看起来清晰了。在 DATE_FORMAT() 和 TIME_FORMAT() 中，第一个参数是待格式化的列，第二个参数是格式代码。顺便说一下，DATE_FORMAT() 可以返回时间，所以其实没必要使用 TIME_FORMAT()。是否使用，取决于编码风格。

之前的写法所出现的问题（例如分钟没有补 0，时和分没有用冒号隔开，以及需要用 `IF()` 判断上下午），在这里都消失了。用格式化代码 `'%l:%i %p'` 即可达到我们想要的效果。如果还想显示秒，那么可以直接用 `'%r'` 来替代那三个代码。表 11-2 展示了各种格式代码及相应的返回值。

表11-2：日期和时间格式代码

代码	描述	结果
`%a`	简写的周几	(Sun...Sat)
`%b`	简写的月份名字	(Jan...Dec)
`%c`	月份（数字形式）	(1...12)
`%d`	一个月中的哪一天（数字形式）	(00...31)
`%D`	一个月中的哪一天（带有英文后缀）	(1st, 2nd, 3rd...)
`%e`	一个月中的哪一天（数字形式）	(0...31)
`%f`	毫秒（数字形式）	(000000...999999)
`%h`	小时	(01...12)
`%H`	小时	(00...23)
`%i`	分钟（数字形式）	(00...59)
`%I`	小时	(01...12)
`%j`	一年中的哪一天	(001...366)
`%k`	小时	(0...23)
`%l`	小时	(1...12)
`%m`	月份（数字形式）	(01...12)
`%M`	月份名字	(January...December)
`%p`	AM 或 PM	AM or PM
`%r`	时间，12 小时制	(hh:mm:ss [AP]M)
`%s`	秒	(00...59)
`%S`	秒	(00...59)
`%T`	时间，24 小时制	(hh:mm:ss)
`%u`	一年中的第几周（以周一为一周的第一天）	(0...52)
`%U`	一年中的第几周（以周日为一周的第一天）	(0...52)
`%v`	一年中的第几周（以周一为一周的第一天，与 %x 一起使用）	(1...53)
`%V`	一年中的第几周（以周日为一周的第一天，与 %X 一起使用）	(1...53)
`%w`	一周中的哪一天	(0=Sunday...6=Saturday)
`%W`	周几	(Sunday...Saturday)
`%x`	某一周所属的年份（以周一为一周的第一天，四位数字形式，与 %v 一起使用）	(yyyy)
`%X`	某一周所属的年份（以周日为一周的第一天，四位数字形式，与 %V 一起使用）	(yyyy)
`%y`	年份（两位数字形式）	(yy)
`%Y`	年份（四位数字形式）	(yyyy)
`%%`	字面的 %	'%'

世界各地都有自己喜好的日期和时间的格式标准。下一节就来研究这个问题，并看看如何调整时区。

11.5 调整格式标准和时区

日期和时间的格式有几种标准。例如一年中的最后一日，可以写成 12-31-2014 或 31-12-2014。要用哪种格式，一般取决于你所在的地区，或你的老板和客户的喜好，或一些其他因素。想查出一个特定标准的日期格式，可以用 GET_FORMAT()：

```
SELECT GET_FORMAT(DATE, 'USA');

+-------------------------+
| GET_FORMAT(DATE, 'USA') |
+-------------------------+
| %m.%d.%Y                |
+-------------------------+
```

顾名思义，GET_FORMAT() 可以返回某一地区的时间格式，该格式能直接用于 DATE_FORMAT()，使时间值格式化。可能你会觉得奇怪，美国竟然用句点而不是用横杠来分隔年月日。GET_FORMAT() 的用法是：在第一个参数填写你想要日期还是时间，或者两个都要（DATE、TIME 或 DATETIME），然后在第二个参数填写日期或时间标准的名称，可选值如下：

- EUR 代表欧洲
- INTERNAL 代表没有标点符号的时间格式
- ISO 代表 ISO 9075 标准
- JIS 代表日本工业标准
- USA 代表美国

MySQL 默认使用 ISO（yyyy-mm-dd hh:mm:ss）来显示日期和时间。

用 GET_FORMAT() 试试以下这个简单的例子：

```
SELECT GET_FORMAT(DATE, 'USA'), GET_FORMAT(TIME, 'USA');

+-------------------------+-------------------------+
| GET_FORMAT(DATE, 'USA') | GET_FORMAT(TIME, 'USA') |
+-------------------------+-------------------------+
| %m.%d.%Y                | %h:%i:%s %p             |
+-------------------------+-------------------------+
```

试试用 GET_FORMAT() 来返回各种标准代码，以便熟悉不同的显示格式——当然，也可以查阅文档（http://dev.mysql.com/doc/refman/5.6/en/date-and-time-functions.html#function_get-format）。试完以后，运行下面这条 SQL 语句，看看该函数如何与 DATE_FORMAT() 结合使用：

```
SELECT DATE_FORMAT(CURDATE(), GET_FORMAT(DATE,'EUR'))
   AS 'Date in Europe',
DATE_FORMAT(CURDATE(), GET_FORMAT(DATE,'USA'))
   AS 'Date in U.S.',
REPLACE(DATE_FORMAT(CURDATE(), GET_FORMAT(DATE,'USA')), '.', '-')
   AS 'Another Date in U.S.';
```

```
+----------------+-------------+---------------------+
| Date in Europe | Date in U.S. | Another Date in U.S. |
+----------------+-------------+---------------------+
| 18.02.2014     | 02.18.2014   | 02-18-2014           |
+----------------+-------------+---------------------+
```

因为我并不认同美国用句点来分隔年月日，所以，我在最后那个域用了 REPLACE()，将句点替换成横杠。GET_FORMAT() 其实并不常用，但不妨稍作了解。比较常用且与之类似的一个函数是 CONVERT_TZ()。

CONVERT_TZ() 可以进行时区转换。不过，在转换之前，我们还是先用以下语句查查服务器当前使用的是哪个时区：

```
SHOW VARIABLES LIKE 'time_zone';

+---------------+--------+
| Variable_name | Value  |
+---------------+--------+
| time_zone     | SYSTEM |
+---------------+--------+
```

这个结果的意思是，我们使用的是文件系统的时间，而这个时间一般按服务器所处的位置来定。假设我们的观鸟网站使用的服务器位于美国马萨诸塞州的波士顿，使用的是美国东部时间。然后，有个意大利罗马的用户在早上（欧洲中部时间）录入了一条鸟种发现信息，但我们不希望它记录波士顿的时间，而记录原时区的时间。否则，美国人会以为意大利人总是在晚上去观鸟，或者总是在日间发现猫头鹰之类的夜行动物。于是，我们需要用 CONVERT_TZ() 来调整一下时间。

CONVERT_TZ() 接受三个参数：日期和时间，来自哪个时区，想转换成哪个时区。下面是一个例子：

```
SELECT common_name AS 'Bird',
CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
DATE_FORMAT(time_seen, '%r') AS 'System Time Spotted',
DATE_FORMAT(CONVERT_TZ(time_seen, 'US/Eastern', 'Europe/Rome'), '%r')
  AS 'Birder Time Spotted'
FROM bird_sightings
JOIN humans USING(human_id)
JOIN rookery.birds USING(bird_id)
JOIN rookery.conservation_status USING(conservation_status_id) LIMIT 3;

+----------------+------------------+-------------------+---------------------+
| Bird           | Birdwatcher      |System Time Spotted| Birder Time Spotted |
+----------------+------------------+-------------------+---------------------+
| Whimbrel       | Richard Stringer | 04:57:12 AM       | 10:57:12 AM         |
| Eskimo Curlew  | Elena Bokova     | 05:09:27 AM       | 11:09:27 AM         |
| Marbled Godwit | Rusty Osborne    | 05:13:25 AM       | 11:13:25 AM         |
+----------------+------------------+-------------------+---------------------+
```

注意，此结果集中，我们转换出了一个比系统时区快六个小时的时区。那是因为我们假设所有人都在罗马所在的时区。其实我们可以给 humans 表加一列，用于填写用户所在时区，

或让他们自定义。至于如何猜测时区，可以在用户注册时，让他们的浏览器告知我们其所在位置，又或者用其他更聪明的方法。同时，我们也允许用户在发现猜测出错时，重新选择时区。不过，无论你怎样确定和保存时区，都要修改上面那条 SQL 语句，把时间改成 CONVERT_TZ() 转换的那个值。

注意，CONVERT_TZ() 所接受的时区代码不一定是三位的（如 CET 代表欧洲中部时间）。具体可选什么时区，都已设置在 MySQL 中（CET 也是这样被设置进去的）。如果你运行上面那条 SQL 语句，却看见 CONVERT_TZ() 返回了 NULL，那可能是因为该时区信息没加载好。如果是 Unix 之类的系统，在安装 MySQL 和 MariaDB 时，你会在 /usr/share/zoneinfo 目录中找到时区文件。列出该目录的文件，就会看到 CONVERT_TZ() 所使用的时区代码。例如，其中的 US 目录就有一个叫 Eastern 的文件。因此，可以使用 US/Eastern 这个时区。若想安装时区文件，可以使用以下命令（文件路径请按实际填写）：

```
mysql_tzinfo_to_sql /usr/share/zoneinfo | mysql -p -u root mysql
```

如果你用的是 Windows，可以到 Oracle 的网站下载时区表（http://dev.mysql.com/downloads/timezones.html）。该网页上会有一些安装指南教你怎样安装。装完之后，再运行一次上面那条 SQL 语句，确保安装正确。

你也可以把时区设置改得与服务器所在的时区不同。不需要移动服务器或调整文件系统的时钟，就可以修改服务器的时区。我们可以给它设置一个更加全球化的时区，比如格林威治标准时间（GMT 或 UTC）。因为观鸟研究这门学问可追溯到英国（这得归功于像约瑟夫 · 班克斯和查尔斯 · 达尔文这样的植物学家），所以，我们就选用 GMT 吧。可以用 SET 语句来设置时区：

```
SET GLOBAL time_zone = 'GMT';
```

如果只想在本次会话应用该时区设置，那就去掉 GLOBAL 选项。如果不想服务器重启后设置都被重置，最好将它写在服务器的配置文件中（即 my.cnf 或 my.ini）。具体的做法是把下面这行代码写进 [mysqld] 那一部分：

```
default-time-zone='GMT'
```

如果不用 SET 而用这种方法，就需要重启服务器才能使它被读取生效。重启之后，再运行一次 SHOW VARIABLES，看看结果如何。

修改服务器时区设置，或者用 CONVERT_TZ() 将显示的时间调成用户的时区，都能使用户产生归属感。没有这些，用户可能会觉得自己被排斥。所以，为了使自己的网站和服务更加国际化，还得好好学习 CONVERT_TZ() 才行。

11.6　日期和时间的加减

MySQL 和 MariaDB 都提供了一些修改指定日期和时间的内置函数。你可以使用它们在某个日期上进行加减运算，使其变成未来或过去的时间。而实现这种运算的最主要、最常用的函数，就是 DATE_ADD() 和 DATE_SUB()。它们的语法一样：第一个参数是要修改的日期，第二个参数是时间量。时间量的写法是：在 INTERVAL 关键字后接上数字和单位（例如

INTERVAL 1 DAY)。

先看一个 DATE_ADD() 的例子。假设我们想把英国所有用户的过期日延迟三个月，可以采用如下方式：

```
UPDATE humans
SET membership_expiration = DATE_ADD(membership_expiration, INTERVAL 3 MONTH)
WHERE country_id = 'uk'
AND membership_expiration > CURDATE( );
```

此例中，我们选取英国的未过期用户，然后给他们的 membership_expiration 加上 3 个月。注意，判断是否已过期是很简单的运算，只要在 WHERE 中用大于号（>）来比较 membership_expiration 和当前日期即可。另外，请注意我们是如何将 membership_expiration 修改成三个月之后的。因为日期和时间函数不会将结果应用到列中，所以需要用其他方法来修改保存好的数据。想知道 DATE_ADD() 接受什么单位，以及一些类似的日期和时间函数，可参考表 11-1。

再看看 DATE_SUB()。假设有个叫 Melissa Lee 的会员想将自己的过期日改成一年之后，但却错改成了两年之后。那么，我们可以通过以下语句进行修正：

```
UPDATE humans
SET membership_expiration = DATE_SUB(membership_expiration, INTERVAL 1 YEAR)
WHERE CONCAT(name_first, SPACE(1), name_last) = 'Melissa Lee';
```

其实，我们应该先查出准确的 human_id，然后在 WHERE 子句中使用它，因为叫 Melissa Lee 的可能不止一个。

DATE_ADD() 非常有用，所以，再多看几个它的用例吧。首先将上例改成不用 DATE_SUB()，而用 DATE_ADD()：

```
UPDATE humans
SET membership_expiration = DATE_ADD(membership_expiration, INTERVAL -1 YEAR)
WHERE CONCAT(name_first, SPACE(1), name_last) = 'Melissa Lee';
```

基本上没改什么，只是换成了 DATE_ADD()，并将数量变成 -1，以代表我们想要减去 1 年，而非增加 1 年（尽管函数名叫 DATE_ADD()）。

再看一个使用 DATE_ADD() 的例子。假设有个会员在记录鸟种发现信息时，不知为何日期和时间都错了。她告知我们，在 time_seen 的记录中，正确的时间应是错误时间的一天两小时之后。于是，我们先找出该次发现的 sighting_id，然后执行以下 SQL 语句来更新日期和时间：

```
UPDATE bird_sightings
SET time_seen = DATE_ADD(time_seen, INTERVAL '1 2' DAY_HOUR)
WHERE sighting_id = 16;
```

此例中，单位是 DAY_HOUR（日和小时），所以声明数量时，要按顺序给出日数和小时数，并把它们放在引号中间。如果你想做的是减法（如一天两小时之前），可以在引号中的其中一个参数前加上负号。顺便说一下，在同一个 DATE_ADD() 中，不能既做加法又做减法。只能先在一个 DATE_ADD() 中进行一种运算，然后把它嵌进做另一种运算的 DATE_ADD() 中。表 11-1 列有各种可用的组合单位。

当使用 DATE_ADD() 之类的函数来让 MySQL 计算新的日期和时间时，它会在后台确定新结果。基本上，它是先将所有日期和时间转换成秒，然后再进行加减，最后返回新的日期和时间。当你想更精确地控制这些运算时，有时候会想确定运算的方法。这个时候可以用 TIME_TO_SEC() 和 SEC_TO_TIME()。

TIME_TO_SEC() 可将时间转换成秒数，这样运算起来很简单。如果你传给它一个含有日期和时间的值，它也只会取时间部分。下面演示一个例子，看看它返回的结果如何：

```
SELECT TIME(NOW()),
TIME_TO_SEC(NOW()),
TIME_TO_SEC(NOW()) / 60 /60 AS 'Hours';

+---------------------+--------------------+------------+
| NOW()               | TIME_TO_SEC(NOW()) | Hours      |
+---------------------+--------------------+------------+
| 2014-02-18 03:30:00 |              12600 | 3.50000000 |
+---------------------+--------------------+------------+
```

第一列是当前的日期和时间。时间部分是 3:30 a.m.。而在第二列，TIME_TO_SEC() 将该时间（一天中的三个半小时）转换成了 12 600 秒。然后，第三列确认了 12 600 秒正是 3.5 小时。

相反，如果已知一件事的持续秒数，就可以用 SEC_TO_TIME() 来计算出时间。假设你想得知两件事之间的间隔时间，例如，我们在网上推出了一个鸟种识别测试。在这个测试中，用户需要从图中认出一种鸟。我们可以用一列记下图片显示的一刻，然后，用同一个表的另一列记录用户输入正确答案的一刻。这样，我们就可以用 SEC_TO_TIME() 来计算出间隔时间（以 hh:mm:ss 形式显示）。为了演示这个例子，我们先建一个记录每个观鸟者测试成绩的表：

```
CREATE TABLE bird_identification_tests
(test_id INT AUTO_INCREMENT KEY,
 human_id INT, bird_id INT,
 id_start TIME,
 id_end TIME);
```

这个表并不复杂：我们只想记录会员的 human_id，图片中鸟种的 bird_id，以及测试的开始时间和结束时间。我们不关心日期，只关心会员花了多长时间才识别出一种鸟。现在，向该表录入数据，因为只想试一下 SEC_TO_TIME()，所以一行数据就够了：

```
INSERT INTO bird_identification_tests
VALUES(NULL, 16, 125, CURTIME(), NULL);
```

注意，我们暂时不录入 id_end，那要等到会员做完测试时才有数据。这只是一种模拟，如果是在真实的应用中，我们会用一个脚本封装这个 INSERT，然后在图片展示时，执行该脚本以录入开始时间。当用户识别出鸟种时，再执行另一个包含 UPDATE 的脚本，以记录结束时间。现在，先继续模拟。执行 INSERT 之后，稍等片刻，再执行以下 SQL 语句，写入 id_end：

```
UPDATE bird_identification_tests
SET id_end = CURTIME();
```

这样，表中唯一一行的 id_end 就被更新为“当前时间”了。现在我们可以在 SELECT 中运

用 SEC_TO_TIME()，看看这个函数是怎么回事：

```
SELECT CONCAT(name_first, SPACE(1), name_last)
   AS 'Birdwatcher',
common_name AS 'Bird',
SEC_TO_TIME( TIME_TO_SEC(id_end) - TIME_TO_SEC(id_start) )
   AS 'Time Elapsed'
FROM bird_identification_tests
JOIN humans USING(human_id)
JOIN rookery.birds USING(bird_id);

+-------------+-----------------+--------------+
| Birdwatcher | Bird            | Time Elapsed |
+-------------+-----------------+--------------+
| Ricky Adams | Crested Shelduck | 00:01:21    |
+-------------+-----------------+--------------+
```

不过，当 id_start 和 id_end 不在同一天时，该 SQL 语句会出现问题。例如，在午夜前开始测试，在午夜后完成测试。这样，id_end 就会小于 id_start，看起来就像是事件在开始前就结束了。为了容纳这种可能，需要构造一个包含 IF() 函数的更复杂的 SQL 语句，以识别这种少有的情况。而对于超过 24 小时才完成测试的情况，则无法识别，你只能用其他方法，例如，在超过 24 小时后，断开会话，以防止记录这样的时间。若真想记录，就需要用 DATETIME 类型，以及能比较日期和时间的函数（这会在下一节讲到）。

现在再多看一个日期加减函数：PERIOD_ADD()。它接受两个参数，第一个是日期，第二个是想要增加的月的数量。也可以用它来减少日期中的月数，只要给第二个参数传负数即可。

PERIOD_ADD() 在本章中看起来有点奇怪，因为它接受的参数不是日期类型，而是字符串类型，同时，返回值也是字符串。这个作为参数的字符串需要含有一个两位或四位的年份，以及一个两位的月份（例如，2014 年 4 月可以写成 1404 或 201404）。现在用 birdwatchers 库的数据来试试这个函数。

假设我们想统计每个会员发现了多少鸟，不过，限定时间范围为上一季度。这看起来好像很简单，似乎只要在 SELECT 的 WHERE 中用 QUARTER() 就能做到，如下所示：

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
COUNT(time_seen) AS 'Sightings Recorded'
FROM bird_sightings
JOIN humans USING(human_id)
WHERE QUARTER(time_seen) = (QUARTER(CURDATE()) - 1)
AND YEAR(time_seen) = (YEAR(CURDATE( )) - 1)
GROUP BY human_id LIMIT 5;

Empty set (0.14 sec)
```

返回的却是空集。这里有两个问题。一，如果你在第一季度执行 QUARTER(CURDATE()) - 1，它并不会智能地返回 4，只会返回 0，而 QUARTER(time_seen) 只可能在 1 到 4 之间，所以在第一季度执行时，单凭这句就会返回空集；二，如果你误以为 QUARTER(CURDATE()) - 1 能从第一季度退回到第四季度，那么当然会想让年份也退回到上一年，所以你会写出 AND YEAR(time_seen) = (YEAR(CURDATE( )) - 1)，但要是你在第二、三、四季度执行，那就真

的退了一整年，这样并不符合“限定时间范围为上一季度”这一需求。

因此得修改这条 SQL 语句。我们需要多次用到 PERIOD_ADD() 以及其他之前讲过的日期时间函数。改好的 SQL 语句如下，不管在哪个季度执行，它都能返回上一季度每个用户所发现的鸟种总数：

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
COUNT(time_seen) AS 'Sightings Recorded'
FROM bird_sightings
JOIN humans USING(human_id)
WHERE CONCAT(QUARTER(time_seen), YEAR(time_seen)) =
CONCAT(
   QUARTER(
      STR_TO_DATE(
         PERIOD_ADD( EXTRACT(YEAR_MONTH FROM CURDATE()), -3),
                             '%Y%m') ),
   YEAR(
      STR_TO_DATE(
         PERIOD_ADD( EXTRACT(YEAR_MONTH FROM CURDATE()), -3),
                             '%Y%m') ) )
GROUP BY human_id LIMIT 5;

+-------------------+--------------------+
| Birdwatcher       | Sightings Recorded |
+-------------------+--------------------+
| Richard Stringer  |                  1 |
| Rusty Osborne     |                  1 |
| Elena Bokova      |                  3 |
| Katerina Smirnova |                  3 |
| Anahit Vanetsyan  |                  1 |
+-------------------+--------------------+
```

为了方便阅读，我加了不少缩进。首先，使用 EXTRACT() 从 CURDATE() 抽取出年份和月份，并转换成 PERIOD_ADD() 所接受的格式（yyyymm）；然后用 PERIOD_ADD() 求得三个月前的月份；接着分别用 QUARTER() 和 YEAR() 求出其季度数和所在年份。其间，我们还用了 STR_TO_DATE() 来将 PERIOD_ADD() 的结果转换成日期。

最后，把季度数和年份拼接起来，与同样拼接的 time_seen 的季度数和年份作比较。如果 EXTRACT() 有 YEAR_QUARTER 选项的话，那会简单得多，不用我们自己计算两次“三个月前的月份”，并分别输入到 QUARTER() 和 YEAR()，再拼接结果。确实，有时我们需要自己来实现一些 MySQL 和 MariaDB 没有提供的选项。不过，官方还是会偶尔补上新功能的。如果没有，那就自己写些复杂的 SQL 语句吧。

11.7　比较日期和时间

我们已经在本书的一些例子中看过几个比较日期和时间的方法。其实这种事情是可以用函数来解决的。其中，最直接的就是 DATEDIFF() 和 TIMEDIFF()。有了它们，你就可以很轻松地比较两个日期或时间。下面来看一些用例。

humans 表有 membership_expiration 这一列，用于记录每个人的会员资格失效日期。假

设我们想在每个人的档案页面提醒他们距离失效日还有多少天，可以像下面这样使用 DATEDIFF()：

```
SELECT CURDATE() AS 'Today',
DATE_FORMAT(membership_expiration, '%M %e, %Y')
   AS 'Date Membership Expires',
DATEDIFF(membership_expiration, CURDATE())
   AS 'Days Until Expiration'
FROM humans
WHERE human_id = 4;

+------------+-------------------------+-----------------------+
| Today      | Date Membership Expires | Days Until Expiration |
+------------+-------------------------+-----------------------+
| 2014-02-13 | September 22, 2013      |                  -144 |
+------------+-------------------------+-----------------------+
```

注意，这里的 DATEDIFF() 返回了一个负数。这是因为 membership_expiration 中的日期早于执行 CURDATE() 那一天的日期。如果你将它们互换，返回值就会是正数。如果你只关心两个日期相差多少天，而不关心哪个更早，那可以再用 ABS() 来将 DATEDIFF() 的结果转成绝对值，这样无论两个日期的顺序如何，结果都是正数。顺便说一下，尽管这个函数可接受 DATE 和 TIME 的组合，但在比较时，TIME 部分是无意义的。

与 DATEDIFF() 类似，TIMEDIFF() 也可用于比较时间。在测试它之前，先建一个使用日期和时间的表。假设我们决定组织和赞助一个观鸟活动，让观鸟爱好者集合起来，去找寻一些有趣的鸟。为了记录相关信息，我们在 birdwatchers 库中建一个 birding_events 表：

```
CREATE TABLE birding_events
(event_id INT AUTO_INCREMENT KEY,
 event_name VARCHAR(255),
 event_description TEXT,
 meeting_point VARCHAR(255),
 event_date DATE,
 start_time TIME);
```

此表中，我们最关心的列是 start_time，即活动的开始时间。现在，我们给 birding_events 增加一个观鸟活动，输入以下 SQL 语句：

```
INSERT INTO birding_events
VALUES (NULL, 'Sandpipers in San Diego',
"Birdwatching Outing in San Diego to look for Sandpipers,
Curlews, Godwits, Snipes and other shore birds.
Birders will walk the beaches and surrounding area in groups of six.
A light lunch will be provided.",
"Hotel del Coronado, the deck near the entrance to the restaurant.",
 '2014-06-15', '09:00:00');
```

现在可以开始试用 TIMEDIFF() 了。输入以下语句，判断活动还有多久才开始：

```
SELECT NOW(), event_date, start_time,
DATEDIFF(event_date, DATE(NOW())) AS 'Days to Event',
TIMEDIFF(start_time, TIME(NOW())) AS 'Time to Start'
FROM birding_events;
```

```
+---------------------+-----------+-----------+------------+---------------+
| NOW()               | event_date | start_time |Days to Event| Time to Start |
+---------------------+-----------+-----------+------------+---------------+
| 2014-02-14 06:45:24 | 2014-06-15 | 09:00:00   |         121 | 02:14:36      |
+---------------------+-----------+-----------+------------+---------------+
```

该活动会在此 SQL 语句执行后的 121 天 2 小时 14 分 36 秒后开始。这个结果是对的，但 Time to Start 所显示的样式看上去就像是一天中的某个时间点，而不像是一种计数。所以，我们决定用 DATE_FORMAT() 使它更美观。另外，也用 CONCAT() 将它和天数放在一起：

```
SELECT NOW(), event_date, start_time,
CONCAT(
   DATEDIFF(event_date, DATE(NOW())), ' Days, ',
   DATE_FORMAT(TIMEDIFF(start_time, TIME(NOW())), '%k hours, %i minutes'))
   AS 'Time to Event'
FROM birding_events;
```

```
+---------------------+-----------+----------+-----------------------------+
| NOW()               | event_date |start_time| Time to Event               |
+---------------------+-----------+----------+-----------------------------+
| 2014-02-14 06:46:25 | 2014-06-15 | 09:00:00 | 121 Days, 2 hours, 13 minutes |
+---------------------+-----------+----------+-----------------------------+
```

要想使这条 SQL 语句正确执行，你得认真检查括号的数量。这里，NOW() 被嵌在 DATE() 和 TIME() 中，而 DATE() 和 TIME() 又分别被嵌在 DATEDIFF() 和 TIMEDIFF() 中，这样，我们才能得出当前日期和时间与 start_time 的差异。然后，用 DATE_FORMAT() 将 TIMEDIFF() 格式化，最后，用 CONCAT() 将两个时间差拼接起来。

试过这个 SQL 语句之后，我们觉得将日期和时间保存在一列中会更简单。而我在本章的第一节说过，我们会介绍如何改变时间列的类型。现在就来看看。再建一列 event_datetime，类型为 DATETIME：

```
ALTER TABLE birding_events
ADD COLUMN event_datetime DATETIME;
```

这一列可用来保存日期和时间。现在将开始日期和开始时间合到一起，输到这列中：

```
UPDATE birding_events
SET event_datetime = CONCAT(event_date,SPACE(1), start_time);
```

CONCAT() 用于将日期和时间拼接起来。然后，MySQL 会自动将拼接结果从字符串转成时间值，再写进 event_datetime。我们来执行一条 SELECT 语句，看看结果如何：

```
SELECT event_date, start_time, event_datetime
FROM birding_events;
```

```
+------------+-----------+---------------------+
| event_date | start_time | event_datetime      |
+------------+-----------+---------------------+
| 2014-06-15 | 09:00:00   | 2014-06-15 09:00:00 |
+------------+-----------+---------------------+
```

更新成功。用这一列再计算一次活动还有多久才开始：

```
SELECT NOW(), event_datetime,
CONCAT(DATEDIFF(event_datetime, NOW() ), ' Days, ',
       TIME_FORMAT( TIMEDIFF( TIME(event_datetime), CURTIME() ),
                    '%k hours, %i minutes') )
   AS 'Time to Event'
FROM birding_events;

+---------------------+---------------------+-------------------------------+
| NOW()               | event_datetime      | Time to Event                 |
+---------------------+---------------------+-------------------------------+
| 2014-02-14 05:48:55 | 2014-06-15 09:00:00 | 121 Days, 3 hours, 11 minutes |
+---------------------+---------------------+-------------------------------+
```

看起来没什么问题，而且，这样比分成两列更好。因为原来的两列已没什么用了，所以，可以将 birding_events 的 event_date 和 start_time 去掉：

```
ALTER TABLE birding_events
DROP COLUMN event_date,
DROP COLUMN start_time;
```

至此，我们已经顺利将分开的两列日期和时间合成一列。你一开始可能就打算建一列而不是两列，正如上面那些例子。但是，你无法保证总能选择正确。因此，我才想带你走一遍这个合并日期和时间的流程，让你具备一定经验。

11.8 小结

本章几乎讲遍了 MySQL 和 MariaDB 中所有的日期和时间函数。剩下没讲的并不多，这其中包括了一些别名（例如，DATE_ADD() 的别名 ADDDATE()，DATE_SUB() 的别名 SUBDATE()），以及一些特殊用途的函数，你可以在需要用到它们的时候再去了解。总之，你所学的知识会让你受用多年。

我们之所以要讲这么多日期和时间函数，主要是因为日期和时间是人们在日常生活中很关注的问题，例如人们总是想知道一件事情何时发生过，发生了多久，或何时才会发生，人们需要约定时间。因此，日期和时间信息就成为了数据库中很重要的一部分。必须熟悉日期和时间函数，以便在解决问题时信手拈来。为了达到这种境界，请做完本章习题，它们会令你对本章内容的印象更加深刻。

11.9 习题

以下习题会用到日期和时间函数，以及第 10 章介绍的一些字符串函数。其中有些还会需要你用 UPDATE 来改变表中的数据。通过使用日期和时间函数来更新数据，你会更好地理解这些函数的能力。可参考第 8 章中 UPDATE 的用法。

(1) 写一条 SQL 语句，筛选出 humans 表中的英国用户。查出姓和名，并拼接起来。另外，也查出注册日和过期日。用 DATE_FORMAT() 将所有日期显示成类似这样的格式：Sun., Feb. 2, 1979。注意不要漏掉标点符号（即缩写词后面的句号和逗号，但年份后不需要句号和逗号）。格式代码可参考表 11-2。

写完后，执行语句并得出结果，检查结果是否正确。若不正确，改到正确为止。

(2) 写一个 SELECT，查出会员的名单和会员资格的过期日，要求结果集按 membership_expiration 列排序。然后使用 UPDATE 修改 membership_expiration。用 ADDDATE() 将过期日延后 1 个月又 15 天，要求只对 2014 年 6 月 30 日后过期的会员操作。日期和时间代码可参考表 11-1。你还需要在 WHERE 中用字符串。写完后，再执行一次刚才的 SELECT，并比较两个结果集，确保只改了该改的人，并且改的日期也正确。

延期后，用 DATESUB() 将那一部分用户的过期日提早 5 天。改完后，再执行一次 SELECT，与上一次 SELECT 的结果对比一下。

再改改过期日，但这次用 ADD_DATE() 来将过期日提早 10 天。注意，你得用负值作参数。改完之后，再执行 SELECT 一次看改得是否正确。

(3) 在 11.6 节中，我们建了一个 bird_identification_tests 表，并录入了一行数据来进行测试。现在，给该表插入至少五行。这些记录需与另外两个 human_id 和其他 bird_id 关联。录入时，像那一节里所做的，用 CURTIME() 填充 id_start 列，而 id_end 列则填 NULL。接着，为了再用 CURTIME() 填充 id_end 列，在每次 INSERT 后都执行一次 UPDATE，这样两列的时间就会不同了。注意每次 INSERT 和 UPDATE 需间隔一定时间。

录入完毕之后，用一个含有 TIMEDIFF() 的 SELECT 查出每条记录的 id_start 和 id_end 的时间间隔。注意 TIMEDIFF() 的参数顺序，使结果不要出现负数。然后，再加入每个会员的名。这需要用到 JOIN（9.2 节介绍过它的用法）。

(4) 再写一条 SELECT，要求查出 birds 表的 common_name，以及 birdwatchers 表的 id_start 和 id_end。使用 TIMEDIFF() 计算两列在时间上的差异。进行表连接时，记住要调整 JOIN，反映两个表在不同的数据库中。写完后执行一下，检查是否正确。然后，加上 GROUP BY，按 bird_id 分组，将 TIMEDIFF() 包在 AVG() 中，以计算平均值。给该值起一个类似 Avg. Time 这样的别名。改完后运行看看结果。其中平均值应该有四位小数（例如 2 分 19 秒显示为 219.0000）。

接着，重写该 SELECT，将四位小数的时间改成 TIME 格式。首先你需要用 TRIM()，带上 TRAILLING 选项和 .0000 字符串，将小数位去掉。然后运行一遍看结果怎样。接着再用 LPAD() 包住它，给左边补 0，使其变成 hhmmss 的格式。之后，再运行一次 SELECT 检查一下。这两个字符串函数在 10.1.3 节中都讲过。

最后，用 STR_TO_DATE 将补 0 后的字符串（如 000219）转换成时间。想拼出 hhmmss 这样的格式代码，可参考表 11-2。如果你只提供时间的格式代码，则 STR_TO_DATE() 只会转换出时间，而这正是我们现在想要的。执行该 SELECT，确保结果无误。若有误，则修改至无误。

(5) 重写上题最后你成功构建的那个 SELECT。将平均值放到 DATE_FORMAT() 中。把格式改成：01 minute(s), 21 seconds。改完后，执行看看。附加题：用一个字符串函数把分钟数和秒数前的 0 去掉。用 IF() 来判断是否需要给 minute 和 second 加 s。

第 12 章

聚合函数和数值函数

数据库经常需要与数字打交道：总有求值、统计或计算的需求。有时，你也可能想将计算结果四舍五入成自己喜欢的样式。为此，MySQL 和 MariaDB 提供了一些数字和算术函数。其中有些还被称为*聚合函数*。本章将会介绍很多数值函数，以及大部分聚合函数，但并不包括那些与统计相关的高级函数，或者那些与微积分和几何相关的数学函数。我们讲的都是最有用和最常用的一些函数，剩下的那些，如果你实在有需要，可自行研究。

12.1 聚合函数

对数据库中的数据进行统计，能得出一些有用的信息。如果数据库中保存了一个组织机构的各种活动信息，那么我们可以对这些信息进行统计分析。例如，如果数据库保存了商品销售情况的相关数据，那么我们可以通过统计来制定一些营销策略。

同理，在 `birdwatchers` 数据库中运用聚合函数，我们就可以了解观鸟网会员的使用习惯，得知他们参与了什么事件，或其他什么活动。而在 `rookery` 库中，则可以得出鸟类的相关信息。这有助于会员在野外找鸟，或者帮助他们了解鸟类的生存状况。我们也可以通过这些数据统计出会员都在哪些地方发现过鸟类。

本节就来看看能求出这些信息的聚合函数。为了将数据集合在一起进行分类统计，我们有时候需要用到 `GROUP BY`。而有些聚合函数，如前几章用于统计表中行数的 `COUNT()`，不需要与 `GROUP BY` 一起使用（至少某些情况是不需要的）。我们就先看看 `COUNT()`，然后再看其他一些简单的统计函数，例如平均值函数。

12.1.1 计数

计数是最简单的计算之一。我们在小时候刚刚接触数学时就会学到。所以，现在就先从计

数函数 COUNT() 开始吧。

假设我们想知道 birds 表含有多少种鸟，可以在 mysql 中输入以下语句：

```
SELECT COUNT(*)
FROM birds;

+----------+
| COUNT(*) |
+----------+
|    28891 |
+----------+
```

注意，这里不需要 GROUP BY。那是因为我们想查出整个表的行数，而不是对表中数据进行分组后的各组的行数——换句话说，整个表就是一组。另外也请注意，我们用了一个星号作为 COUNT() 的参数。星号是通配符，指示 MySQL 统计所有查出的行。因为该 SQL 语句没有加 WHERE，所以选择的行就是整个表的行。

有不少鸟种没有俗名，所以在 birds 表中，它们的 common_name 为空。而 COUNT() 有这样一个规定：如果参数是列名而非星号，则它只会对非 NULL 的行进行计数。我们可以改改数据，看看不同的参数会有怎样的结果。输入以下两条 SQL 语句：

```
UPDATE birds
SET common_name = NULL
WHERE common_name = '';

SELECT COUNT(common_name)
FROM birds;

+--------------------+
| COUNT(common_name) |
+--------------------+
|               9553 |
+--------------------+
```

这就是表中有俗名的鸟的数目。其实不用修改数据，用 WHERE 也可以得出这一结果。那样就需要限定只查出 common_name 不等于 '' 的行。不过，我们已经将所有 '' 更新成 NULL，所以，现在只能根据 NULL 来写 WHERE 了：

```
SELECT COUNT(*) FROM birds
WHERE common_name IS NULL;

+----------+
| COUNT(*) |
+----------+
|    19338 |
+----------+
```

数目不同了，因为我们计算的是 common_name 为 NULL 的行——用了 IS NULL 运算符。而之前计算的是 common_name 非 NULL 的行。要算出这些非 NULL 的行，需要将 WHERE 按如下修改：

```
SELECT COUNT(*) FROM birds
WHERE common_name IS NOT NULL;

+----------+
| COUNT(*) |
+----------+
|     9553 |
+----------+
```

只需改成 IS NOT NULL，就与之前一样了。

COUNT() 还有其他不错的用法，现在就来玩一玩。试试统计每科的鸟种数量。这需要用到 GROUP BY，SQL 语句如下：

```
SELECT COUNT(*)
FROM birds
GROUP BY family_id;

+----------+
| COUNT(*) |
+----------+
|        5 |
|        6 |
|      248 |
|      119 |
|      168 |
|       39 |
|      223 |
|      ... |
+----------+

227 rows in set (0.15 sec)
```

此例中，我们告诉 MySQL 要按 family_id 进行 GROUP BY 操作。所以，它就按 family_id 对所有行进行分组，然后统计每一组的行数。因为结果会有 227 行，所以我省略了一些，以节省空间。这条 SQL 语句运行正常，但出来的结果没什么用。如果结果集中还有科名，那就好了。于是，我们用 JOIN 来连接 bird_families 表：

```
SELECT bird_families.scientific_name AS 'Bird Family',
COUNT(*) AS 'Number of Species'
FROM birds JOIN bird_families USING(family_id)
GROUP BY birds.family_id;

+--------------------+-------------------+
| Bird Family        | Number of Species |
+--------------------+-------------------+
| Gaviidae           |                 6 |
| Anatidae           |               248 |
| Charadriidae       |               119 |
| Laridae            |               168 |
| Sternidae          |                39 |
| Caprimulgidae      |               223 |
| Sittidae           |                92 |
| ...                |                   |
+--------------------+-------------------+
```

```
225 rows in set (0.17 sec)
```

这看起来好多了，结果也更有意思了。同样，我省略了部分结果。不过，你要注意，这次总共只有 225 行。那是因为 birds 表中有些行的 family_id 是 NULL。在使用数据库的过程中，要留意这些差异。不要因为你不是在排查问题就忽略它们，事实上，这就是一个问题。

现在来修改 SELECT，使其返回在 bird_families 中找不到匹配记录的行数。这需要使用 LEFT JOIN（我们在 9.2.1 节中举过例子，这里再讲一次）：

```
SELECT bird_families.scientific_name AS 'Bird Family',
COUNT(*) AS 'Number of Species'
FROM birds LEFT JOIN bird_families USING(family_id)
GROUP BY birds.family_id;

+--------------------+-------------------+
| Bird Family        | Number of Species |
+--------------------+-------------------+
| NULL               |                 4 |
| NULL               |                 1 |
| Gaviidae           |                 6 |
| Anatidae           |               248 |
| Charadriidae       |               119 |
| Laridae            |               168 |
| Sternidae          |                39 |
| Caprimulgidae      |               223 |
| Sittidae           |                92 |
| ...                |                   |
+--------------------+-------------------+

225 rows in set (0.17 sec)
```

从结果可以看出，birds 表中有些行的 family_id 为 NULL，有一行的 family_id 在 bird_families 中没有记录。虽然我们可以再写一个 SELECT 来查出到底是哪一行，但这偏离了学习聚合函数的目的。所以，就让我们假设这个问题已经解决，并回到聚合函数的话题上。

你可能已经发现，在前两个例子的结果中，科名没有按字母序显示。这是因为，GROUP BY 是根据它所带的列名来排序的（这两个例子用的都是 family_id）。如果想按科名（bird_families 表的 scientific_name）来排，就需要改动 GROUP BY 子句。试试输入以下语句：

```
SELECT bird_families.scientific_name AS 'Bird Family',
COUNT(*) AS 'Number of Species'
FROM birds LEFT JOIN bird_families USING(family_id)
GROUP BY bird_families.scientific_name;

+--------------------+-------------------+
| Bird Family        | Number of Species |
+--------------------+-------------------+
| Acanthisittidae    |                 9 |
| Acanthizidae       |               238 |
| Accipitridae       |               481 |
| Acrocephalidae     |               122 |
```

```
| Aegithalidae       |                 49 |
| Aegithinidae       |                 20 |
| Aegothelidae       |                 21 |
| Alaudidae          |                447 |
| ...                |                    |
+--------------------+--------------------+
```

这已经好多了，但如果能在结果集的底部显示记录的总数，就更好了。可以用另一条 SQL 语句来求总数，而如果想在同一条 SQL 语句中求出，则可在 `GROUP BY` 中加上 `WITH ROLLUP`：

```
SELECT bird_families.scientific_name AS 'Bird Family',
COUNT(*) AS 'Number of Species'
FROM birds JOIN bird_families USING(family_id)
GROUP BY bird_families.scientific_name WITH ROLLUP;

+--------------------+--------------------+
| Bird Family        | Number of Species |
+--------------------+--------------------+
| Acanthisittidae    |                  9 |
| Acanthizidae       |                238 |
| Accipitridae       |                481 |
| Acrocephalidae     |                122 |
| Aegithalidae       |                 49 |
| Aegithinidae       |                 20 |
| Aegothelidae       |                 21 |
| Alaudidae          |                447 |
| ...                |                    |
| NULL               |              28891 |
+--------------------+--------------------+
```

这样，结果集的最后一行就显示了记录的总数，它也确实与本节的第一个例子的结果一样。该行第一个域的 NULL 不是指它的 `family_id` 不匹配，而是因为它是所有组的总数，没有特定的组名，所以 MySQL 就使用了 NULL。不过，我们可以自己做些调整，使得它有值。在修改的同时，顺便也统计每个目所含鸟种的数量：

```
SELECT IFNULL( bird_orders.scientific_name, '') AS 'Bird Order',
IFNULL( bird_families.scientific_name, 'Total:') AS 'Bird Family',
COUNT(*) AS 'Number of Species'
FROM birds
JOIN bird_families USING(family_id)
JOIN bird_orders USING(order_id)
GROUP BY bird_orders.scientific_name, bird_families.scientific_name
WITH ROLLUP;

+---------------------+--------------------+--------------------+
| Bird Order          | Bird Family        | Number of Species |
+---------------------+--------------------+--------------------+
| Anseriformes        | Anhimidae          |                  3 |
| Anseriformes        | Total:             |                  3 |
| Apodiformes         | Apodidae           |                316 |
| Apodiformes         | Hemiprocnidae      |                 16 |
| Apodiformes         | Trochilidae        |                809 |
```

```
| Apodiformes         | Total:              |              1141 |
| Caprimulgiformes    | Aegothelidae        |                21 |
| Caprimulgiformes    | Caprimulgidae       |               224 |
| Caprimulgiformes    | Nyctibiidae         |                17 |
| Caprimulgiformes    | Podargidae          |                26 |
| ...                 |                     |                   |
|                     | Total:              |             28890 |
+---------------------+---------------------+-------------------+
```

除了增加一个域来显示每个目中鸟种的数量，我们还使用了 `IFNULL()` 来处理目名和科名。这个函数会告诉 MySQL，如果该域返回 NULL 值，则替换成我们另外给的值或字符串——在这里，可能出现 NULL 的就是统计数以外的列。因为我们的 SQL 语句先按科、后按目来做统计，所以结果是对的。

此语句的效果虽然不算很厉害，却能方便地与脚本组合，在网页上显示查询结果。可以用应用编程接口来检查总数的值：在第二个域，然后调整它。也可以用应用编程接口的脚本来做这些简单的计算，而不是让 MySQL 来做。但有时候，在数据库系统层做会比较好。这是因为，我常发现这样会更加紧凑、更易于维护。好了，说这么多已经够了，我们来继续学习 `COUNT()` 以外的聚合函数。

12.1.2 对一组数据进行运算

我们在第 11 章建过一个叫 `bird_identification_tests` 的表，用于记录会员在网上做的“鸟类识别测试”的情况。现假设我们想告知会员他们测试所花时间的平均值。一种简单的算法就是，把每次的时间（用 `id_end` 减去 `id_start`）加总，然后除以测试次数。而要实现加总，可以用 `SUM()`。

不过，在继续之前，先看看其中一个会员的测试记录，再决定怎样做。我们使用 `TIMEDIFF()` 来比较测试的开始时间和结束时间（这在 11.7 节中介绍过）：

```
SELECT common_name AS 'Bird',
TIME_TO_SEC( TIMEDIFF(id_end, id_start) )
AS 'Seconds to Identify'
FROM bird_identification_tests
JOIN humans USING(human_id)
JOIN rookery.birds USING(bird_id)
WHERE name_first = 'Ricky' AND name_last = 'Adams';

+--------------------+---------------------+
| Bird               | Seconds to Identify |
+--------------------+---------------------+
| Crested Shelduck   |                  81 |
| Moluccan Scrubfowl |                 174 |
| Indian Pond-Heron  |                 181 |
+--------------------+---------------------+
```

因为需要先将每次测试的时间加总，再求平均值，所以我们得用 `TIME_TO_SEC()` 把 `TIMEDIFF()` 的结果转换成秒数，如把 121（即 1 分 21 秒）转换成 81 秒。接着，再多做一步，以更好地理解时间函数和 `SUM()` 的用途：

```
SELECT CONCAT(name_first, SPACE(1), name_last)
   AS 'Birdwatcher',
SUM(TIME_TO_SEC( TIMEDIFF(id_end, id_start) ) )
   AS 'Total Seconds for Identifications'
FROM bird_identification_tests
JOIN humans USING(human_id)
JOIN rookery.birds USING(bird_id)
WHERE name_first = 'Ricky' AND name_last = 'Adams';

+-------------+-----------------------------------+
| Birdwatcher | Total Seconds for Identifications |
+-------------+-----------------------------------+
| Ricky Adams |                               436 |
+-------------+-----------------------------------+
```

这就是 Ricky Adams 三次测试的总秒数。注意，SUM() 也是一个不必与 GROUP BY 一起使用的聚合函数。接着，再改改 SQL 语句，以计算平均值（将 436 秒除以 3）。我们试试用两个子查询，分别查出 436 和 3。这种写法复杂而且低效，不用跟着做，看看就好：

```
SELECT Identifications, Seconds,
(Seconds / Identifications) AS 'Avg. Seconds/Identification'
FROM
  ( SELECT human_id, COUNT(*) AS 'Identifications'
    FROM bird_identification_tests
    JOIN humans USING(human_id)
    JOIN rookery.birds USING(bird_id)
    WHERE name_first = 'Ricky' AND name_last = 'Adams')
       AS row_count
  JOIN
  ( SELECT human_id, CONCAT(name_first, SPACE(1), name_last)
       AS 'Birdwatcher',
    SUM(TIME_TO_SEC(TIMEDIFF(id_end, id_start)))
       AS 'Seconds'
    FROM bird_identification_tests
    JOIN humans USING(human_id)
    JOIN rookery.birds USING(bird_id) )
       AS second_count
  USING(human_id);

+-----------------+---------+-----------------------------+
| Identifications | Seconds | Avg. Seconds/Identification |
+-----------------+---------+-----------------------------+
|               3 |     436 |                    145.3333 |
+-----------------+---------+-----------------------------+
```

这写起来很费劲，应该有更简单的办法才对——确实是有的。那就是使用 AVG()，现在改用它试试：

```
SELECT CONCAT(name_first, SPACE(1), name_last)
   AS 'Birdwatcher',
AVG( TIME_TO_SEC( TIMEDIFF(id_end, id_start)) )
   AS 'Avg. Seconds per Identification'
FROM bird_identification_tests
JOIN humans USING(human_id)
```

```
JOIN rookery.birds USING(bird_id)
WHERE name_first = 'Ricky' AND name_last = 'Adams';

+-------------+--------------------------------+
| Birdwatcher | Avg. Seconds per Identification |
+-------------+--------------------------------+
| Ricky Adams |                       145.3333 |
+-------------+--------------------------------+
```

这样就简单得多了，而且不需要用到子查询。若再去掉 WHERE，就可以查出所有会员的平均值。下面就来改改，并将时间格式改成分钟和秒，同时也将测试次数显示出来。时间格式的更改，会用到 SEC_TO_TIME()，它会将 TIME_TO_SEC() 和 AVG() 算出的平均秒数变回时间格式：

```
SELECT CONCAT(name_first, SPACE(1), name_last)
   AS 'Birdwatcher',
COUNT(*) AS 'Birds',
TIME_FORMAT(
     SEC_TO_TIME(AVG( TIME_TO_SEC( TIMEDIFF(id_end, id_start)))),
     '%i:%s' )
   AS 'Avg. Time'
FROM bird_identification_tests
JOIN humans USING(human_id)
JOIN rookery.birds USING(bird_id)
GROUP BY human_id LIMIT 3;

+---------------+-------+-----------+
| Birdwatcher   | Birds | Avg. Time |
+---------------+-------+-----------+
| Rusty Osborne |     2 | 01:59     |
| Lexi Hollar   |     3 | 00:23     |
| Ricky Adams   |     3 | 02:25     |
+---------------+-------+-----------+
```

这次我们查了三个会员。另外，也查出他们各自所做的测试次数。而且，平均时间的格式也更美观了。从结果可以看出，Ricky Adams 的平均时间比 Lexi Hollar 的长得多。这可能是因为 Lexi Hollar 确实做得比较快，又或者是 Ricky Adams 做的时候分神了吧。

因为使用了 LIMIT，所以看不出谁的平均时间最长，以及谁做得最快。如果想知道的话，可以去掉 LIMIT，然后将整条 SQL 语句作为另一条 SQL 语句的子查询，并在那一条 SQL 语句中使用 ORDER BY。也就是说，使内部的 SQL 语句返回所有会员的平均时间列表，然后用外部的 SQL 语句将该列表按我们想要的顺序来排列：

```
SELECT Birdwatcher, avg_time AS 'Avg. Time'
FROM
  (SELECT CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
   COUNT(*) AS 'Birds',
   TIME_FORMAT( SEC_TO_TIME( AVG(
                  TIME_TO_SEC( TIMEDIFF(id_end, id_start)))
                  ),'%i:%s' ) AS 'avg_time'
    FROM bird_identification_tests
    JOIN humans USING(human_id)
    JOIN rookery.birds USING(bird_id)
```

```
    GROUP BY human_id) AS average_times
ORDER BY avg_time;

+-------------------+-----------+
| Birdwatcher       | Avg. Time |
+-------------------+-----------+
| Lexi Hollar       | 00:23     |
| Geoffrey Dyer     | 00:25     |
| Katerina Smirnova | 00:48     |
| Rusty Osborne     | 01:59     |
| Ricky Adams       | 02:25     |
| Anahit Vanetsyan  | 03:20     |
+-------------------+-----------+
```

这样就能看到，Lexi Hollar 是最快的，而 Anahit Vanetsyan 是最慢的。因为一般无法把 GROUP BY 和 ORDER BY 放在同一条 SQL 语句中，所以我们只能用一个子查询。

如果不需要知道具体的人名，那么可以直接用 MAX() 和 MIN()。现在就来用这两个聚合函数修改上例：

```
SELECT MIN(avg_time) AS 'Minimum Avg. Time',
MAX(avg_time) AS 'Maximum Avg. Time'
FROM humans
JOIN
 (SELECT human_id, COUNT(*) AS 'Birds',
  TIME_FORMAT(
     SEC_TO_TIME( AVG(
        TIME_TO_SEC( TIMEDIFF(id_end, id_start)))
           ), '%i:%s' ) AS 'avg_time'
  FROM bird_identification_tests
  JOIN humans USING(human_id)
  JOIN rookery.birds USING(bird_id)
  GROUP BY human_id ) AS average_times;

+-------------------+-------------------+
| Minimum Avg. Time | Maximum Avg. Time |
+-------------------+-------------------+
| 00:23             | 03:20             |
+-------------------+-------------------+
```

与上例的结果比较，可知这次也是正确的。如果想知道每个用户的最长和最短完成时间，而不是平均时间，可以使用如下语句：

```
SELECT CONCAT(name_first, SPACE(1), name_last) AS 'Birdwatcher',
TIME_FORMAT(SEC_TO_TIME(
              MIN(TIME_TO_SEC( TIMEDIFF(id_end, id_start)))
              ),%i:%s' ) AS 'Minimum Time',
TIME_FORMAT(SEC_TO_TIME(
              MAX(TIME_TO_SEC( TIMEDIFF(id_end, id_start)))
              ), '%i:%s' ) AS 'Maximum Time'
FROM bird_identification_tests
JOIN humans USING(human_id)
JOIN rookery.birds USING(bird_id)
GROUP BY Birdwatcher;
```

```
+-------------------+--------------+--------------+
| Birdwatcher       | Minimum Time | Maximum Time |
+-------------------+--------------+--------------+
| Anahit Vanetsyan  | 00:20        | 08:48        |
| Geoffrey Dyer     | 00:09        | 00:42        |
| Katerina Smirnova | 00:22        | 01:02        |
| Lexi Hollar       | 00:11        | 00:39        |
| Ricky Adams       | 01:21        | 03:01        |
| Rusty Osborne     | 01:50        | 02:08        |
+-------------------+--------------+--------------+
```

这条 SQL 语句显示了按字母排序的会员名字，以及他们各自的最长和最短完成时间。一般来说，只要按会员来分组，就可以对每个会员的记录使用 AVG() 和 MAX() 之类的聚合函数。这里，我们没用 COUNT()。

其实玩法还有很多。我们可以查询识别哪种鸟花的时间最长或最短。这样就可以知道哪些鸟是最难认的，并将它们留给资深会员去做。我们甚至可以考虑，那些平均时间很短的会员，是不是因为他们没有抽中难认的鸟。如果想排除这些记录，可以使用 STDDEV() 和 VARIANCE() 之类的聚合函数来做一些高级的统计分析。不过对于新手来说，这些应该不太需要掌握，你现在只需要知道它们的存在即可。

在进入下一个话题之前，再看一个不涉及时间值的 MIN() 和 MAX() 的用例。bird_sightings 表存有每次会员发现鸟类的相关信息，其中的 location_gps 列是发现地的 GPS 坐标。它包含两个 11 位的数字：纬度和经度。因为有些鸟会在南北之间迁徙，所以我们想查出每种鸟的最南和最北的发现地。可以使用 SUBSTRING() 来截取出纬度，然后用 MAX() 得到最北的纬度，用 MIN() 得到最南的纬度。代码如下：

```
SELECT common_name AS 'Bird',
MAX(SUBSTRING(location_gps, 1, 11)) AS 'Furthest North',
MIN(SUBSTRING(location_gps, 1, 11)) AS 'Furthest South'
FROM birdwatchers.bird_sightings
JOIN rookery.birds USING(bird_id)
WHERE location_gps IS NOT NULL
GROUP BY bird_id LIMIT 3;

+-----------------+----------------+----------------+
| Bird            | Furthest North | Furthest South |
+-----------------+----------------+----------------+
| Eskimo Curlew   | 66.16051056    | 66.16051056    |
| Whimbrel        | 30.29138551    | 30.29138551    |
| Eurasian Curlew | 51.70469364    | 42.69096856    |
+-----------------+----------------+----------------+
```

此结果集中，因为前两种鸟只被发现过一次，所以最南和最北都是同一个地点，而 Eurasian Curlew 则很明显是在不同地方都被见到过。

12.1.3 拼接同组的值

在结束聚合函数这个话题之前，我还想介绍一个函数，那就是 GROUP_CONCAT()。它并不常

用，但在某些特殊情况下，它会特别有用。它可以把一个组所有的值拼接成一个以逗号分隔的串。如果没有它的话，你就需要写子查询，并在其中使用 CONCAT_WS() 来做拼接。

举个例子，若要列出某个鸟目中的所有鸟科，很简单，一个 SELECT 就可以做到。但若是想同时列出目和科，并要求结果集中，同属一目的所有科都在一行中呢？如果没有 GROUP_CONCAT()，会很麻烦。下面就来看看它能产生怎样的效果。输入以下语句：

```
SELECT bird_orders.scientific_name AS 'Bird Order',
GROUP_CONCAT(bird_families.scientific_name)
AS 'Bird Families in Order'
FROM rookery.bird_families
JOIN rookery.bird_orders USING(order_id)
WHERE bird_orders.scientific_name = 'Charadriiformes'
GROUP BY order_id \G

*************************** 1. row ***************************
           Bird Order: Charadriiformes
Bird Families in Order:

Charadriidae,Laridae,Sternidae,Burhinidae,Chionidae,Pluvianellidae,
Dromadidae,Haematopodidae,Ibidorhynchidae,Recurvirostridae,
Jacanidae,Scolopacidae,Turnicidae,Glareolidae,Pedionomidae,
Thinocoridae,Rostratulidae,Stercorariidae,Alcidae
```

为了节省空间，我只查了一个目。如果想查所有目，只需去掉 WHERE：

```
SELECT bird_orders.scientific_name AS 'Bird Order',
GROUP_CONCAT(bird_families.scientific_name SEPARATOR ', ')
AS 'Bird Families in Order'
FROM rookery.bird_families
JOIN rookery.bird_orders USING(order_id)
GROUP BY order_id \G
```

试试上面的语句，你会发现，GROUP_CONCAT() 中的 SEPARATOR 子句使每个科名之间都以逗号和空格分隔开来了。

12.2 数值函数

能以某种方式改变数字的函数叫作*数值函数*。它们本身是不做计算的。如果做的话，那就叫算术函数了。它们能帮你简化结果集中的数字，例如四舍五入或者求绝对值。这些用数值函数轻易就能完成。本节讲的就是这些函数。

12.2.1 四舍五入

计算机很精确，所以有时它们的计算结果会带有很多位小数。这对你来说可能不算什么问题，尤其是在你把数字只作为其他函数的参数，而不是要展示出来时。但一般来说，人们总是比较习惯于经过舍入的数字，而不需要像计算机那样精确。对于这种情况，可以使用一些数值函数来进行四舍五入。

在 5.2 节中，我们在 MariaDB 中建了一些含有动态列的表，用来保存与用户观鸟偏好

相关的调研信息。现在就用它们来测试一些数值函数。如果你没建这些表，或者没用 MariaDB，那么是做不了以下例子的。

作为开始，先回顾一下当时写过的一条 SQL 语句。先从我的网站导入更多数据，然后再运行一次：

```
SELECT IFNULL(COLUMN_GET(choices, answer AS CHAR), 'total')
AS 'Birding Site', COUNT(*) AS 'Votes'
FROM survey_answers
JOIN survey_questions USING(question_id)
WHERE survey_id = 1
AND question_id = 1
GROUP BY answer WITH ROLLUP;

+--------------+-------+
| Birding Site | Votes |
+--------------+-------+
| forest       |    30 |
| shore        |    42 |
| backyard     |    14 |
| total        |    86 |
+--------------+-------+
```

这个结果集显示了观鸟地点偏好的投票情况。让我们来计算百分比。首先，我们要知道总票数。这可以用子查询来返回，但为了做得更简单，我们先用一个 `SELECT` 查出它，并把它临时保存在一个用户自定义变量之中。用户自定义变量是临时性的，它只存在于当前用户的会话中，也只能由其创建者访问。创建这样一个用户自定义变量需要用 `SET` 语句，然后指定一个以 `@` 开头的名字，再用 `=` 连接上一个值，或者一个表达式，又或者一条返回单个值的 SQL 语句。现在就来给我们的例子建一个用户自定义变量。在你的 MariaDB 中，输入以下语句：

```
SET @fav_site_total =
(SELECT COUNT(*)
FROM survey_answers
JOIN survey_questions USING(question_id)
WHERE survey_id = 1
AND question_id = 1);

SELECT @fav_site_total;

+-----------------+
| @fav_site_total |
+-----------------+
|              86 |
+-----------------+
```

因为我后来还导了更多数据到 `survey_answers` 表，所以数值比之前的高了。在下一个例子中，就可以看到这个总数是正确的。现在用这个变量来计算每个选项投票的百分比：

```
SELECT COLUMN_GET(choices, answer AS CHAR)
  AS 'Birding Site',
COUNT(*) AS 'Votes',
```

```
(COUNT(*) / @fav_site_total) AS 'Percent'
FROM survey_answers
JOIN survey_questions USING(question_id)
WHERE survey_id = 1
AND question_id = 1
GROUP BY answer;

+--------------+-------+---------+
| Birding Site | Votes | Percent |
+--------------+-------+---------+
| forest       |    30 |  0.3488 |
| shore        |    42 |  0.4884 |
| backyard     |    14 |  0.1628 |
+--------------+-------+---------+
```

此语句用每种选项的票数除以总票数（保存在变量中的那个），得出的结果有四位小数。让我们把它乘以 100，使其变成百分数的形式，然后再用 ROUND() 来消除小数部分。最后，再用 CONCAT() 加上百分号：

```
SELECT COLUMN_GET(choices, answer AS CHAR)
  AS 'Birding Site',
COUNT(*) AS 'Votes',
CONCAT( ROUND( (COUNT(*) / @fav_site_total) * 100), '%')
  AS 'Percent'
FROM survey_answers
JOIN survey_questions USING(question_id)
WHERE survey_id = 1
AND question_id = 1
GROUP BY answer;

+--------------+-------+---------+
| Birding Site | Votes | Percent |
+--------------+-------+---------+
| forest       |    30 | 35%     |
| shore        |    42 | 49%     |
| backyard     |    14 | 16%     |
+--------------+-------+---------+
```

注意，这里 ROUND() 使得前两个数上舍入，而最后一个数下舍入。这是因为四舍五入就是这样。下面我们再改改，让它保留一位小数：

```
SELECT COLUMN_GET(choices, answer AS CHAR)
AS 'Birding Site',
COUNT(*) AS 'Votes',
CONCAT( ROUND( (COUNT(*) / @fav_site_total) * 100, 1), '%') AS 'Percent'
FROM survey_answers
JOIN survey_questions USING(question_id)
WHERE survey_id = 1
AND question_id = 1
GROUP BY answer;

+--------------+-------+---------+
| Birding Site | Votes | Percent |
```

```
+--------------+-------+---------+
| forest       |    30 | 34.9%   |
| shore        |    42 | 48.8%   |
| backyard     |    14 | 16.3%   |
+--------------+-------+---------+
```

在保留小数的情况下，舍入依然正确。假设我们保守一点，希望无论如何都上舍入，又或者无论如何都下舍入，那就需要别的函数了。

12.2.2 上舍入或下舍入

若只想下舍入，用 FLOOR()。若只想上舍入，则用 CEILING()。现在再拿上一个例子，演示下舍入：

```
SELECT COLUMN_GET(choices, answer AS CHAR)
  AS 'Birding Site',
COUNT(*) AS 'Votes',
CONCAT( FLOOR( (COUNT(*) / @fav_site_total) * 100), '%')
  AS 'Percent'
FROM survey_answers
JOIN survey_questions USING(question_id)
WHERE survey_id = 1
AND question_id = 1
GROUP BY answer;

+--------------+-------+---------+
| Birding Site | Votes | Percent |
+--------------+-------+---------+
| forest       |    30 | 34%     |
| shore        |    42 | 48%     |
| backyard     |    14 | 16%     |
+--------------+-------+---------+
```

这次，我们把 ROUND() 换成了 FLOOR()，使其无论如何都下舍入。不过 FLOOR() 不能指定保留多少位小数，它只会返回整数。

如果想上舍入，可以用 CEILING()，如下所示：

```
SELECT COLUMN_GET(choices, answer AS CHAR)
AS 'Birding Site',
COUNT(*) AS 'Votes',
CONCAT( CEILING( (COUNT(*) / @fav_site_total) * 100), '%') AS 'Percent'
FROM survey_answers
JOIN survey_questions USING(question_id)
WHERE survey_id = 1
AND question_id = 1
GROUP BY answer;

+--------------+-------+---------+
| Birding Site | Votes | Percent |
+--------------+-------+---------+
| forest       |    30 | 35%     |
| shore        |    42 | 49%     |
| backyard     |    14 | 17%     |
+--------------+-------+---------+
```

这就全都上舍入了。如果参数不含小数，则不会有任何改变。

12.2.3 截短数字

如果你并不特别想要上舍入或下舍入，只想去掉小数，那可以用 TRUNCATE()。试试将它用在刚才的例子上：

```
SELECT COLUMN_GET(choices, answer AS CHAR)
  AS 'Birding Site',
COUNT(*) AS 'Votes',
CONCAT( TRUNCATE( (COUNT(*) / @fav_site_total) * 100, 1), '%')
  AS 'Percent'
FROM survey_answers
JOIN survey_questions USING(question_id)
WHERE survey_id = 1
AND question_id = 1
GROUP BY answer;

+--------------+-------+---------+
| Birding Site | Votes | Percent |
+--------------+-------+---------+
| forest       |    30 | 34.8%   |
| shore        |    42 | 48.8%   |
| backyard     |    14 | 16.2%   |
+--------------+-------+---------+
```

顾名思义，这个函数就是把指定的小数位（本例中是 1）之后的小数给截掉。

12.2.4 消除负数

有时候，用函数处理数字时，你可能会把参数顺序搞错，导致结果带有负号。如果你只想知道两个数字之间差多少，可以使用 ABS() 来求绝对值，使结果不带负号。事实上，绝对值在某些数学计算中很重要。

为了演示该函数的用法，我会用到前面“鸟类识别”的一些例子。我们计算每个用户测试时间的总和，但这次不需要按 human_id 来分组：

```
SELECT
SUM( TIME_TO_SEC( TIMEDIFF(id_start, id_end) ) )
   AS 'Total Seconds for All',
ABS( SUM( TIME_TO_SEC( TIMEDIFF(id_start, id_end) ) ) )
   AS 'Absolute Total'
FROM bird_identification_tests;

+-----------------------+----------------+
| Total Seconds for All | Absolute Total |
+-----------------------+----------------+
|                 -1689 |           1689 |
+-----------------------+----------------+
```

这个函数和例子并不复杂。结果中第一个域之所以是负数，是因为在 TIMEDIFF() 中，我们将 id_start 放在 id_end 前面了。当然你可以把它们的位置反过来，以计算出正数，但有

些时候，你不知道两个参数哪个比较大。这时，可以用 ABS() 来解决。

如果想知道一个值是正还是负，可用 SIGN()。当传给它的参数为正时，它会返回 1，为负时，则返回 -1，参数为 0，则返回 0。

我们再用鸟类识别的数据来演示一下。现假设我们想知道哪些鸟的测试所用时间比平均时间少。因为最少平均时间的计算已在 12.1.2 节中做过，所以这次我们就重用其中部分语句，并且将计算结果保存在用户自定义变量中，再用该变量与 bird_identification_tests 表的每一行进行比较，这样就可以只查出测试所用时间比平均时间少的行。下面，创建这个变量，然后在服务器上测试它：

```
SET @min_avg_time =
(SELECT MIN(avg_time) FROM
  (SELECT AVG( TIME_TO_SEC( TIMEDIFF(id_end, id_start)))
     AS 'avg_time'
   FROM bird_identification_tests
   GROUP BY human_id) AS average_times);

SELECT @min_avg_time;

+---------------+
| @min_avg_time |
+---------------+
|       23.6667 |
+---------------+
```

这是正确的。之前我们算出过 23 秒，但那次是用 TIME_FORMAT() 四舍五入过的，而这次是精确值。接着，在 WHERE 中，用这个变量配合 SIGN() 来做一些数值比较。输入以下语句：

```
SELECT CONCAT(name_first, SPACE(1), name_last)
   AS 'Birdwatcher',
common_name AS 'Bird',
ROUND(@min_avg_time - TIME_TO_SEC( TIMEDIFF(id_end, id_start) ) )
   AS 'Seconds Less than Average'
FROM bird_identification_tests
JOIN humans USING(human_id)
JOIN rookery.birds USING(bird_id)
WHERE SIGN( TIME_TO_SEC( TIMEDIFF(id_end, id_start) - @min_avg_time)) = -1;

+-------------------+---------------------+---------------------------+
| Birdwatcher       | Bird Identified     | Seconds Less than Average |
+-------------------+---------------------+---------------------------+
| Lexi Hollar       | Blue Duck           |                         3 |
| Lexi Hollar       | Trinidad Piping-Guan |                       13 |
| Geoffrey Dyer     | Javan Plover        |                        15 |
| Katerina Smirnova | Blue Duck           |                         2 |
| Anahit Vanetsyan  | Great Crested Grebe |                         4 |
+-------------------+---------------------+---------------------------+
```

这样，我们就在 WHERE 中用 SIGN() 筛选出了测试时间少于平均时间的会员。这个函数的效果很难用其他方法模拟。

12.3 小结

虽然我们没有把所有聚合函数和数值函数都介绍完，但已经把大部分都讲过一遍，其中包含了最常用的那些。被省略的主要是统计分析函数以及算术函数，但它们是很易懂的或专用的（例如，`POWER(2, 8)` 会返回 2 的 8 次方，即 256；`PI()` 会返回 π，即 3.141593）。其实最重要的是，你应熟悉聚合函数和 `GROUP BY`（因为它们很常用），并且掌握本章介绍过的数值函数。除此之外的数值函数，你可能在某些情况下也会用到。如果想了解这些函数，可以查看 MySQL 文档（http://dev.mysql.com/doc/refman/5.6/en/group-by-functions.html）或 MariaDB 文档（https://mariadb.com/kb/en/mariadb/functions-and-modifiers-for-use-with-group-by/）。

12.4 习题

数值函数很简单，一学就会。本章有关数值函数的小节应该都不会让你觉得有难度。而聚合函数就可能有点麻烦。所以，下面有些习题要求你使用数值函数，而大部分都包含聚合函数的练习。其中还有些会需要你将两者组合起来使用。这些习题会使你巩固本章所学知识。本章的习题并不多，你应该能很快完成。

(1) 写一个简单的 `SELECT`，统计 `birds` 表中有多少行的 `common_name` 含有 `Least`。执行它并确保能正常运行。接着，将其改成统计 `birds` 表中有多少行的 `common_name` 含有 `Great`。你会在 `WHERE` 中用到 `LIKE`。

(2) 在 12.1.2 节中，我们谈过如何分组计数。用 `GROUP BY` 将上题的两条 SQL 语句合二为一，使得结果集中有一个域显示 `Least` 鸟的数目，另一个域显示 `Great` 鸟的数目。

(3) 在做这一题时，你可能需要参考 12.1.1 节中统计所有鸟和统计每科鸟的例子。

写一个查询 `birds` 表的 `SELECT`，结果集返回三个域：鸟科名字、鸟科所含鸟种的数量，以及占鸟种总数的百分比。鸟种的总数要让 MySQL 算出来，不要自己写到 SQL 语句中。

执行成功之后，用一个数值函数来修改这条 SQL 语句，让百分比的那个域保留一位小数。

(4) 重做上题。这次另写一个 `SELECT`，只取得鸟种总数。用 `SET` 创建一个用户自定义变量，来保存 MySQL 通过 `SELECT` 取得的值。变量名字自定。

现在修改上题中的 `SELECT`，使用该变量来统计百分比。执行成功后，退出 `mysql` 客户端并重新登录。

再把创建用户自定义变量和使用变量来统计百分比的语句执行一次。看看统计百分比的语句花了多少时间。然后再执行一次上题那个不用变量统计的语句。比较两种方法的运行时长。

(5) `humans` 表中有个 `membership_expiration` 列，里面含有日期值。现使用一个 `SELECT`，查询每个会员的 `membership_expiration` 与 2014-01-01 之间隔了多少个月。如果你不清楚怎么做，可参考 11.7 节。然后，在 `WHERE` 中用 `SIGN()` 检查会员资格是否已过期。只筛选出未过期的用户。做法可参考 12.2.4 节。还有，记得在 `WHERE` 中用 `IF NOT NULL` 将不用付费维持会员资格的人排除掉（即那些没有过期日的）。为表示间隔了多少个月的那个域起名为 `Months to Expiration`。

(6) 修改上题的 SQL 语句，这次不排除资格过期的会员，只排除不用付费维持会员资格的人。用 `CONCAT()` 给会员资格过期的人加上 `" - expired"`。你需要用 `IF()` 来判断哪些人的会员资格已过期，并用 `ABS()` 去掉负号。

(7) 在上题 SQL 语句的基础上，写一条新的 SQL 语句，计算需要付费维持会员资格的人“还有多少个月过期”的平均值，以及过期者“过期了多少个月”的平均值。并且，将 2014-01-01 当作当前日期。你会用到 `AVG()` 来计算平均值。运行成功后，再加两个域，使用 `MIN()`、`MAX()` 和 `GROUP BY`，计算出最少和最多的月数。

第五部分

数据库管理

最后这个部分将探讨 MySQL 和 MariaDB 的一些管理活动。它们不一定与数据库开发相关，但与数据管理相关。其中有些活动是日常的，有些是偶尔的。另外，我们还会讲 MySQL 和 MariaDB 以外的一些内容。

首先，第 13 章会讲到用户账号和权限的管理。我们在本书开头简单介绍过这一话题，而本章会进行更深入的探讨。我们会讲解如何更精确地指定每个用户对每个库和每个表的权限。

第 14 章将讨论如何备份数据库。这是很重要的管理任务。除此之外，我们还会提及不太常做的恢复备份。不过这一旦做起来，通常是因为数据出现了严重、紧急的问题。我一如既往地建议你做完每章结尾的习题。而因为本章的话题如此重要，所以相关习题尤其不容忽视。

第 15 章讲解导入大量数据的管理任务。你可能不会经常从其他数据库或其他文件（例如电子表格或逗号分隔值文件）导入大量数据。但了解导入过程是非常有用的，可以让你在必要时节省时间，减少失败的概率。

本书最后一章，即第 16 章，将简要介绍几个 API，里面含有使用 PHP 和其他编程语言连接并查询 MySQL 和 MariaDB 的例子。几乎所有数据库都需要与 API 打交道，因为使用 API 可以提高数据库操作的灵活性和安全性，也使用户无需直接面对数据库。

第 13 章

用户账号和权限

在本章之前，我们已有几次提到用户账号和权限，而本章将详细讨论这个重要的话题。既然安全性在与数据相关的活动中如此重要，可能有人会觉得应该在本书的开头就详细地讲解这个话题。但我认为，在花很多时间进行权限管理和安全管理这种无聊的任务之前，先学习数据操作会更有趣。而且，在深入了解表和数据库中的其他组件之后，你才更易理解用户权限的重要性，以及思考设置权限的各种方法。所以，现在可以开始考虑用户账号等相关问题，而且你会理解为何我把这些内容放在这里，而不是放在本书开头。

我们会先看看创建用户账号和授予权限的基本知识，然后详细讨论如何限制访问，以及如何给不同的数据库组件授予不同的权限。在掌握限制访问的这些方法后，我们会探讨应该给一些常见的管理账号授予什么权限。最后，我们将学习如何回收权限、删除账号，以及如何修改密码和重命名账号。

13.1 用户账号的基础知识

在本书中，我多次用了*用户账号*这个术语，而不是*用户*。这样做是为了区分“用户名与主机的组合”和“MySQL 或 MariaDB 的使用者”这两个不同的概念。

举个例子，你可以让 root 用户在所有数据库中都拥有最高权限，但限制它只能通过 `localhost` 登录，而不允许远程登录（例如通过互联网登录）。如果允许的话，可能会造成安全问题。简单来说，访问和权限基于用户和主机的组合，这个组合也叫用户账号。

作为 root 用户，你可以用 `CREATE USER` 语句来创建用户账号。以下例子使用此语句来给名为 Lena Stankoska 的一位女性创建用户账号：

```
CREATE USER 'lena_stankoska';
```

此例只创建了用户账号，而没有授予它任何权限。如果想查看一个用户账号有什么权限，可以使用 SHOW GRANTS 语句，如下所示：

```
SHOW GRANTS FOR 'lena_stankoska';

+-------------------------------------------+
| Grants for lena_stankoska@%               |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'lena_stankoska'@'%' |
+-------------------------------------------+
```

注意，结果就像是一条 SQL 语句。除了使用 CREATE USER 来授权，也可以使用如上的 GRANT 语句来做。现在我们将结果拆开来看，但是从右往左看。

其中，用户名是 lena_stankoska，主机是通配符 %。之所以用通配符，是因为我们在 CREATE USER 时没有指定主机。于是，从任何主机登录这个账号都会拥有通用的权限。但这样是不好的。无论何时你都应该指定主机。作为一个初始的例子，我们就在这里用 localhost 吧。至于如何设置主机，我们会在下一节再讲。

结果中的 *.* 是指授权使用所有数据库和所有表（句点前的部分是数据库，句点后的是表）。为了限制用户只能操作某个特定的数据库或表，应把 *.* 改成 database.table。至于如何写，我们很快会讲到。

创建完用户账号之后，一般你就会给它授权。如果想让某个用户账号从 localhost 登录时能使用所有 SQL 语句，可以使用如下 GRANT 语句：

```
GRANT ALL ON rookery.*
TO 'lena_stankoska'@'localhost';

SHOW GRANTS FOR 'lena_stankoska'@'localhost';

+-------------------------------------------------------------------+
| Grants for lena_stankoska@localhost                               |
+-------------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'lena_stankoska'@'localhost'                |
| GRANT ALL PRIVILEGES ON `rookery`.* TO 'lena_stankoska'@'localhost' |
+-------------------------------------------------------------------+
```

注意，现在用户账号 lena_stankoska@localhost 的 SHOW GRANTS 结果变成两行了：第一行与之前的结果相似，而第二行是我们刚刚才授权的。现在除了给别人授权，它还可以对 rookery 数据库进行各种操作。而关于授权的能力以及其他可施加的权限，我们会在本章后面谈到。

因为我们还没给该用户账号指定密码，所以它无需密码便可登录。这样很危险，即相当于任何人不输密码都可以访问该服务器，并能对数据库执行几乎所有的命令。因为我们创建该用户账号只是为了演示如何授予和查看权限，所以先删掉它，之后再重建一次。

删除用户账号的语句是 DROP USER。但在删 Lena Stankoska 这个用户账号之前，其实要考虑很多因素。当执行 CREATE USER 而没指定主机时，我们建了一个带有通配符的用户账号。当我们用 GRANT 来给同一个用户授权时（不过这次是与 localhost 主机的组合），就创建了

另一个用户账号。想理解得更深入，来看看 mysql 数据库的 user 表，那里保存着该用户账号的信息。输入以下语句：

```
SELECT User, Host
FROM mysql.user
WHERE User LIKE 'lena_stankoska';

+----------------+-----------+
| User           | Host      |
+----------------+-----------+
| lena_stankoska | %         |
| lena_stankoska | localhost |
+----------------+-----------+
```

如你所见，虽然我们只想创建一个用户账号，但结果建了两个。如果你之前没搞懂“用户”和“用户账号”的区别，我希望你能从这里看出来。

> 尽管可以直接在 mysql 数据库中访问用户账号的权限，但还是不要用那种方法来修改用户账号。虽然到现在示例都很简单，但某些权限管理的操作其实会影响到 mysql 数据库的多个表。如果不使用本章介绍的一些合适的用户账号语句，而尝试用 INSERT、UPDATE 或 DELETE 来插入、更新或删除 user 表中的用户账号，那就可能会达不到想要的效果，并在其他表中产生孤立数据。

要删除我们给 Lena Stankoska 创建的两个用户账号，需要执行两次 DROP USER，如下所示：

```
DROP USER 'lena_stankoska'@'localhost';
DROP USER 'lena_stankoska'@'%';
```

这就把两个用户账号都删掉了。在接下来的几节中，我们会给她创建更多的用户账号，但会更密切地关注如何限制用户账号的访问权限，以使她不能在未填密码的情况下就登录或随便访问各种资源。

13.2 限制用户账号的访问权限

数据库管理员可以授予用户从任何地方访问并操作所有资源的权限，也可以限制用户只能从某些地方登录，或只能操作某些资源。简而言之，可以限制基于用户名和主机的权限，用户账号访问的数据库组件（如表），以及在数据库组件上会用到的 SQL 语句和函数。本节将介绍这些限制。

13.2.1 用户名和主机

创建用户账号的时候，你得考虑是谁需要登录，以及从哪里登录。现在先看看“谁”的定义。“谁”可以是一个人，也可以是一群人。你可以给每个人都建一个用户名，例如根据实名来起数据库用户名，给 Lena Stankoska 建一个 lena_stankoska 用户名。你也可以给一群人建一个用户名，例如给销售部建一个 sales_dept 用户名。你甚至可以按功能或用途来起名。在这种情况下，一个人可能会有多个用户账号。

假设 Lena Stankoska 是 `rookery` 和 `birdwatchers` 的数据库管理员，那么她就可能拥有多个用户名（假设全都只能从 localhost 登录），例如，lena_stankoska 用于个人作业；admin_backup 用于备份数据；admin_restore 用于恢复备份。而如果她平时还导入大批数据，那么还可以建一个 admin_import。

现在先创建 Lena Stankoska 的个人作业账号，稍后再创建管理员账号。个人账号有两个，名字都叫 lena_stankoska：一个用于 localhost 登录，另一个用于远程登录。并且让她在 localhost 登录时，权限大些，而远程登录时（在家里登录，假设她家有个固定 IP），权限小些。具体来说，账号命名为 lena_stankoska@localhost 和 lena_stankoska@lena_stankoska_home。

定义账号时，主机名可以是能通过 DNS 查找到的名字，也可以是实际的 IP 地址。DNS 查找可以通过外部的域名服务器，也可以是内部的 bind 程序和 hosts 文件（如 Linux 的 /etc/hosts 文件）。如果选择后者，则需要重启 MySQL 才能采用。

下面就是实际操作。输入以下语句，给 Lena Stankoska 创建两个用于个人作业的用户账号：

```
CREATE USER 'lena_stankoska'@'localhost'
IDENTIFIED BY 'her_password_123';

GRANT USAGE ON *.* TO 'lena_stankoska'@'lena_stankoska_home'
IDENTIFIED BY 'her_password_123';
```

此例分别用了 `CREATE USER` 和 `GRANT` 来创建用户账号。如果 `GRANT` 所指的用户名不存在，则会自动创建用户——记住，用户名与主机的组合是一个独特的用户账号。不过，建议你还是先使用 `CREATE USER`，后使用 `GRANT`。另外，我们在每条语句中都加了 `IDENTIFIED BY` 子句，这样给每个用户账号都设置了密码。

接着，我们看看其中一个用户账号的样子。输入以下命令：

```
SHOW GRANTS FOR 'lena_stankoska'@'localhost' \G

*************************** 1. row ***************************
Grants for admin_backup@localhost:
     GRANT USAGE ON *.* TO 'lena_stankoska'@'localhost'
     IDENTIFIED BY PASSWORD ' *B1A8D5415ACE5AB4BBAC120EC1D17766B8EFF1A1'
```

注意，显示出来的密码是加密过的。MySQL 不会让你查看到原始的密码串，也不提供解密的方法。而且，注意在加密的密码前，还多了 `PASSWORD` 关键字。如果你不想像上例那样明文地设置密码，可以在另一台计算机上，先用 `PASSWORD()` 函数进行加密，然后用 `GRANT` 把加密的结果复制到服务器上，如下所示：

```
SELECT PASSWORD('her_password_123');

+-------------------------------------------+
| PASSWORD('its_password_123')              |
+-------------------------------------------+
| *B1A8D5415ACE5AB4BBAC120EC1D17766B8EFF1A1 |
+-------------------------------------------+
```

结果与之前 `SHOW GRANTS` 看到的一样。如果你的服务器在记录所有命令，那么你可能想用

这种方法在自己的个人计算机上给密码加密，以防别人看到你的密码明文。顺便说一下，从 MySQL 5.6 版本开始，包含 PASSWORD 的语句不会被记录。

现在，Lena Stankoska 可以选择某个用户账号来登录了，其中一个只允许她从家里登录，另外四个只允许她从服务器登录。但她还无法访问任何数据库，除了默认的两个（即 test 和 information_schema）。她可以在 test 中进行各种各样的操作，包括建表，还有选择、更新和删除数据。而其他数据库对于她来说则不可访问，甚至不可见。另外，她也无法创建数据库。使用这些用户账号，是很受约束的。下一节我们会更多地了解用户账号可以访问什么资源，并试试让 Lena Stankoska 访问 test 以外的数据库。

13.2.2 SQL权限

除了访问数据库以外，Lena Stankoska 还需要更多权限才能完成她的工作。我们得授予她执行各种任务的权限，例如读写 rookery 和 birdwatchers 数据库的数据。现在，先让 lena_stankoska@localhost 在这两个数据库中有执行 SELECT、INSERT 和 UPDATE 的权限。为了授予用户账号多种权限，我们要将这些权限以逗号分隔的方式，在 GRANT 中列出：

```
GRANT SELECT, INSERT, UPDATE ON rookery.*
TO 'lena_stankoska'@'localhost';

GRANT SELECT, INSERT, UPDATE ON birdwatchers.*
TO 'lena_stankoska'@'localhost';

SHOW GRANTS FOR 'lena_stankoska'@localhost \G

*************************** 1. row ***************************
Grants for lena_stankoska@localhost:
GRANT USAGE ON *.*
TO 'lena_stankoska'@'localhost'

*************************** 2. row ***************************
Grants for lena_stankoska@localhost:
GRANT SELECT, INSERT, UPDATE ON `birdwatchers`.*
TO 'lena_stankoska'@'localhost'

*************************** 3. row ***************************
Grants for lena_stankoska@localhost:
GRANT SELECT, INSERT, UPDATE ON `rookery`.*
TO 'lena_stankoska'@'localhost'
```

有些权限的授予，会同时开放多种 SQL 语句的使用。想知道到底有多少种权限，可参考表 13-1。

虽然 lena_stankoska@localhost 已有足够的权限在这两个数据库中修改数据，但还不能删除数据。当你想给某个用户账号增加权限时，无需再次列出原有的权限，只需在 GRANT 中写上新增的权限即可。如下：

```
GRANT DELETE ON rookery.*
TO 'lena_stankoska'@'localhost';

GRANT DELETE ON birdwatchers.*
```

```
TO 'lena_stankoska'@'localhost';

SHOW GRANTS FOR 'lena_stankoska'@localhost \G

*************************** 1. row ***************************
Grants for lena_stankoska@localhost:
GRANT USAGE ON *.*
TO 'lena_stankoska'@'localhost'

*************************** 2. row ***************************
Grants for lena_stankoska@localhost:
GRANT SELECT, INSERT, UPDATE, DELETE ON `birdwatchers`.*
TO 'lena_stankoska'@'localhost'

*************************** 3. row ***************************
Grants for lena_stankoska@localhost:
GRANT SELECT, INSERT, UPDATE, DELETE ON `rookery`.*
TO 'lena_stankoska'@'localhost'
```

这样，Lena Stankoska 就可以在两个数据库里进行增删改查了，不过这仅限于从 localhost 登录。而从家里登录的话，她还是什么都做不了。我们会在后面再对 lena_stankoska@lena_stankoska_home 进行授权。

表13-1：GRANT和REVOKE语句中可用的权限

权　　限	描　　述
ALL [PRIVILEGES]	授予所有基本的权限，但不包括 GRANT OPTION
ALTER	允许使用 ALTER TABLE 语句，但得先有 CREATE 和 INSERT 权限。如果想重命名一个表，还得先有 DROP 权限。这里有一个安全隐患：用户可以通过对表改名来获得访问权
ALTER ROUTINE	允许用户账号更改或删除存储过程，即允许使用 ALTER FUNCTION、ALTER PROCEDURE、DROP FUNCTION 和 DROP PROCEDURE 语句
CREATE	允许使用 CREATE TABLE 语句。如果想定义索引，还需要 INDEX 权限
CREATE ROUTINE	允许用户账号创建存储过程，即允许使用 CREATE FUNCTION 和 CREATE PROCEDURE 语句。并且，用户账号对自己创建的过程拥有 ALTER ROUTINE 权限
CREATE TEMPORARY TABLES	允许使用 CREATE TEMPORARY TABLES 语句
CREATE USER	允许用户账号执行以下用户账号管理语句：CREATE USER、RENAME USER、REVOKE ALL PRIVILEGES 和 DROP USER
CREATE VIEW	允许使用 CREATE VIEW 语句
DELETE	允许使用 DELETE 语句
DROP	允许使用 DROP TABLE 和 TRUNCATE 语句
EVENT	允许用户账号为事件调度器创建事件，即允许使用 CREATE EVENT、ALTER EVENT 和 DROP EVENT 语句
EXECUTE	允许执行存储过程，即允许使用 EXECUTE 语句
FILE	允许使用 SELECT...INTO OUTFILE 和 LOAD DATA INFILE 语句来导出数据到文件系统，或从文件系统导入数据。这有一个安全隐患。可通过 secure_file_priv 变量来限制指定目录

（续）

权　　限	描　　述
INDEX	允许使用 CREATE INDEX 和 DROP INDEX 语句
INSERT	允许使用 INSERT 语句。这是执行 ANALYZE TABLE、OPTIMIZE TABLE 和 REPAIR TABLE 语句的前提条件
LOCK TABLES	允许使用 LOCK TABLES 语句，但用户必须先对表有 SELECT 权限
PROCESS	允许使用 SHOW PROCESSLIST 和 SHOW ENGINE 语句
RELOAD	允许使用 FLUSH 语句
REPLICATION CLIENT	允许用户查询主从服务器的状态信息，即允许使用 SHOW MASTER STATUS、SHOW SLAVE STATUS 和 SHOW BINARY LOGS 语句
REPLICATION SLAVE	进行从服务器复制时，需要此权限，它将允许读取主服务器的二进制日志事件
SELECT	允许使用 SELECT 语句
SHOW DATABASES	允许使用 SHOW DATABASES 语句来查看所有数据库，包括那些没有权限的数据库
SHOW VIEW	允许使用 SHOW CREATE VIEW 语句
SHUTDOWN	允许使用 mysqladmin 工具的 shutdown 选项
SUPER	允许使用 CHANGE MASTER TO、KILL、PURGE BINARY LOGS、SET GLOBAL 语句，以及 mysqladmin 工具的 debug 选项
TRIGGER	允许用户账号创建或删除触发器，即允许使用 CREATE TRIGGER 和 DROP TRIGGER 语句
UPDATE	允许使用 UPDATE 语句
USAGE	可用于创建无权限的用户，或在不影响用户现有权限的情况下修改其某方面的属性

13.2.3　数据库组件和权限

现在我们来看看用户账号可以访问数据库的哪些部分。你可以允许用户账号访问服务器上所有的数据库，也可以让它只能访问特定的数据库，特定的表，甚至特定的列。下面我们先看看如何指定用户账号可以访问的数据库，之后再看如何指定表和列。

> Lena Stankoska 在家登录时所受的限制比从 localhost 登录时要多。当然，如果她真的想在从家里登录时能访问更多信息，可以先用 ssh 登录操作系统，然后再以 lena_stankoska@localhost 的方式登录 MySQL。这样也不错。给 lena_stankoska@lena_stankoska_home 加上这些限制，可以保证敏感数据不会未经加密就泄露出去。而在操作系统的级别上，可以控制得更多，例如限制使用 ssh 连接，以提高安全性。

1. 限制访问特定的数据库

为了限制 lena_stankoska@lena_stankoska_home 只能在 rookery 上操作，可以采用如下方式：

```
GRANT USAGE ON rookery.*
TO 'lena_stankoska'@'lena_stankoska_home'
```

```
IDENTIFIED BY 'her_password_123';

SHOW GRANTS FOR 'lena_stankoska'@'lena_stankoska_home' \G

*************************** 1. row ***************************
Grants for lena_stankoska@lena_stankoska_home:
    GRANT USAGE ON *.* TO 'lena_stankoska'@'lena_stankoska_home'
    IDENTIFIED BY PASSWORD '*B1A8D5415ACE5AB4BBAC120EC1D17766B8EFF1A1'
```

我们给该用户账号授予了对 rookery 的 USAGE 权限。但是，SHOW GRANTS 的结果没有改变，Lena Stankoska 依然对所有的数据库和表都有 USAGE 权限。如果她从家里登录，并使用列出数据库的命令，会看到以下结果：

```
mysql --user lena_stankoska --password='her_password_123' \
      --host rookery.eu --execute='SHOW DATABASES'

+--------------------+
| Database           |
+--------------------+
| information_schema |
| test               |
+--------------------+
```

她依然看不到 rookery 数据库。这是因为她在该数据库中什么都不能做，甚至不能执行 SHOW TABLES 或 SELECT。要让她看到 rookery，需要给她更多的权限，而不只是 USAGE。我们先给她在 rookery 中执行 SELECT 的权限：

```
GRANT SELECT ON rookery.*
TO 'lena_stankoska'@'lena_stankoska_home';

SHOW GRANTS FOR 'lena_stankoska'@'lena_stankoska_home';

+----------------------------------------------------------------------------+
| Grants for lena_stankoska@lena_stankoska_home                              |
+----------------------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'lena_stankoska'@'lena_stankoska_home'               |
| IDENTIFIED BY PASSWORD '...'                                               |
| GRANT SELECT ON `rookery`.* TO 'lena_stankoska'@'lena_stankoska_home'      |
+----------------------------------------------------------------------------+
```

因为在 GRANT 中不能只给出数据库名，还得写上表名，所以这里用了 .* 来代表 rookery 的所有表。

注意，在这次结果中，用户账号对所有库和表的 USAGE 权限依然存在。而该条目的下一条，便是关于 rookery 的权限。另外，为了适应书页的大小，我把密码部分换成了省略号。现在，Lena Stankoska 可以从家里登录，并在 rookery 中执行 SELECT 了。以下是她从家里登录并执行 SHOW DATABASES 和 SELECT 查询 Avocet 鸟的结果：

```
mysql --user lena_stankoska --password='her_password_123' --host rookery.eu \
      --execute="SHOW DATABASES; \
                 SELECT common_name AS 'Avocets'
                 FROM rookery.birds \
                 WHERE common_name LIKE '%Avocet%';"
```

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| rookery            |
| test               |
+--------------------+
+---------------------+
| Avocets             |
+---------------------+
| Pied Avocet         |
| Red-necked Avocet   |
| Andean Avocet       |
| American Avocet     |
| Mountain Avocetbill |
+---------------------+
```

2. 限制访问特定的表

至此，Lena Stankoska 可以从办公室操作 rookery 和 birdwatchers 这两个数据库。然而，虽然她在家里能对 rookery 进行 SELECT 操作，但不能访问 birdwatchers。现在，我们准许她在家里对 birdwatchers 中的某些表进行 SELECT 操作。

如果我们只打算开放 bird_sightings 的 SELECT 权限，那么可以采用如下方式：

```
GRANT SELECT ON birdwatchers.bird_sightings
TO 'lena_stankoska'@'lena_stankoska_home';

SHOW GRANTS FOR 'lena_stankoska'@'lena_stankoska_home';

+---------------------------------------------------------------------------+
| Grants for lena_stankoska@lena_stankoska_home                             |
+---------------------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'lena_stankoska'@'lena_stankoska_home'              |
| IDENTIFIED BY PASSWORD '...'                                              |
| GRANT SELECT ON `rookery`.* TO 'lena_stankoska'@'lena_stankoska_home'     |
| GRANT SELECT ON `birdwatchers`.`bird_sightings`                           |
| TO 'lena_stankoska'@'lena_stankoska_home'                                 |
+---------------------------------------------------------------------------+
```

这样，bird_sightings 就成为了 Lena Stankoska 唯一能在 birdwatchers 中看到的表。如果她从家里的计算机输入以下命令，便会看到如下结果：

```
mysql --user lena_stankoska --password='her_password_123' --host rookery.eu \
      --execute="SHOW TABLES FROM birdwatchers;"

+------------------------+
| Tables_in_birdwatchers |
+------------------------+
| bird_sightings         |
+------------------------+
```

要想让她访问 birdwatchers 的其他表，可以针对每个表执行 GRANT 语句。如果一个数据库有很多表，这样做就很麻烦。但这确实无可避免。不过，在写这一章时，我已经提了要

求，让 MariaDB 可以一次针对多个表进行 GRANT 操作。所以，说不定未来在 MariaDB 中执行 GRANT 时，你可以轻松一点。暂时来说，你只能手动逐个写 GRANT，或创建一个简短的脚本来帮你操作。

举个例子，假设我们现在想批准 Lena Stankoska 操作 birdwatchers 中所有的表（除了那些包含个人数据和敏感数据的表）。也就是说，批准她操作 humans、birder_families 和 birding_events_children 以外的表。可以写一个如下的 shell 脚本：

```
#!/bin/sh

mysql_connect="mysql --user root -pmy_pwd"

results=`$mysql_connect --skip-column-names \
         --execute 'SHOW TABLES FROM birdwatchers;'`

items=$(echo $results | tr " " "\n")

for item in $items
do

  if [ $item = 'humans' ] ||
     [ $item = 'birder_families' ] ||
     [ $item = 'birding_events_children' ]
  then
    continue
  fi

  `$mysql_connect --execute "GRANT SELECT ON birdwatchers.$item \
                             TO 'lena_stankoska'@'lena_stankoska_home'"`
done

exit
```

这个简单的 shell 脚本用 SHOW TABLES 取得一堆表名，然后对这堆中的每个表名逐个执行 GRANT 语句（但略过了上述三个包含敏感信息的表）。

这样，Lena Stankoska 便可以在办公室处理工作，并在家里检查数据。除此之外，她还可以做一些管理任务，例如备份数据或导入大量数据。做这些任务时，她会用到我们给她创建的那三个管理员账号中的一个。我们会在后面设置这三个账号的权限，这样 Lena Stankoska 就可以完成她需要做的任务。

3. 限制访问特定的列

要想限制用户账号只能访问某些列，需要在 GRANT 的权限关键字后，把允许访问的列以逗号分隔的方式列于括号之中。你看一个具体的例子就肯定能明白。如果同时还授予很多种权限，那这个 GRANT 看起来应该会超级长。

在上一节中，出于安全考虑，我们没给 Lena Stankoska 从家里访问 humans 表的权限。假设我们改变了想法：除了用户的联系方式（例如电子邮件地址），其他的都可以让她访问。为了能与其他表连接，她需要有 human_id 列的访问权限。此外，formal_title、name_first、name_last 和 membership_type 都应该开放。而剩下的列，不是包含敏感信息，就

是对她的工作没用。

使用如下 GRANT 语句：

```
GRANT SELECT (human_id, formal_title, name_first,
name_last, membership_type)
ON birdwatchers.humans
TO 'lena_stankoska'@'lena_stankoska_home';
```

这样，Lena Stankoska 便能在家中访问 humans 表中每个会员的姓名及其会员资格的类型了。

13.3 管理员账号

之前我说过，我们要给 Lena Stankoska 创建三个从 localhost 登录的管理员账号，让她作为数据库管理员在执行任务时使用。这三个账号是 admin_backup、admin_restore 和 admin_import。这些管理员账号很常见，你可能需要创建和使用它们。在第 14 章（讲备份与恢复数据）和第 15 章（讲批量导入数据）中，我们就会在示例中用到这三个账号。本节就来创建它们，再加一个用于授权的账号，并看看需要给它们分配什么权限。

13.3.1 用于备份的用户账号

用户账号 admin_backup 和实用程序 mysqldump 一起使用，可以对 rookery 和 birdwatchers 进行备份。这会在第 14 章介绍。该账号需要的权限不多。

- 最起码，需要有 SELECT 的权限才能读取这两个数据库。应该限制管理员账号只能访问需要备份的库。尤其是，管理员账号不应该拥有对 mysql 数据库的 SELECT 权限，因为该数据库含有登录密码。
- 因为备份时需要锁住表，所以需要 LOCK TABLES 权限。
- 如果数据库含有视图或触发器（本书不涉及这两个话题），那用户账号还分别需要 SHOW VIEW 和 TRIGGER 权限。

基于以上考虑，我们来创建 admin_backup@localhost，并授予它在 rookery 和 birdwatchers 上进行 SELECT 和 LOCK TABLES 的权限。命令如下：

```
CREATE USER 'admin_backup'@'localhost'
IDENTIFIED BY 'its_password_123';

GRANT SELECT, LOCK TABLES
ON rookery.*
TO 'admin_backup'@'localhost';

GRANT SELECT, LOCK TABLES
ON birdwatchers.*
TO 'admin_backup'@'localhost';
```

这样，Lena Stankoska 就可以用 admin_backup 来备份数据库了。而要恢复备份，则用 admin_restore。下面我们就来给它设置权限。

13.3.2 用于恢复备份的用户账号

虽然可以创建一个既做备份又做恢复的账号，但你可能想用单独的两个账号来完成这些任务。这主要是因为备份工作多数是用脚本自动执行的，而恢复数据通常是人工操作的，并会改动甚至破坏服务器上的数据。你应该不会希望能修改数据的账号密码暴露在脚本文件之中。对于本章的例子来说，我们给 admin_restore@localhost 以下权限，用于恢复数据。

- 将数据从 dump 文件恢复到表中时，用户账号至少需要 `INSERT` 权限。
- 在插入数据时，还需要 `LOCK TABLES` 权限来锁住表。
- 需要 `CREATE` 和 `INDEX` 权限来分别创建表和索引。
- 因为 dump 文件可能包含设置校对集的语句，所以需要 `ALTER` 权限。
- 基于 Lena Stankoska 用来恢复表的方法，她可能还想将数据恢复到临时表。这样的话，则需要 `CREATE TEMPORARY TABLES` 权限。（临时表会在连接关闭时被删掉。）
- 如果数据库有视图或触发器，则需要 `CREATE VIEW` 和 `TRIGGER` 权限。

对于我们的数据库使用来说，除了 `CREATE VIEW` 和 `TRIGGER`，其他的权限都是要有的。下面就创建 admin_restore@localhost，并授予它必要的权限：

```
CREATE USER 'admin_restore'@'localhost'
IDENTIFIED BY 'different_pwd_456';

GRANT INSERT, LOCK TABLES, CREATE,
CREATE TEMPORARY TABLES, INDEX, ALTER
ON rookery.*
TO 'admin_restore'@'localhost';

GRANT INSERT, LOCK TABLES, CREATE,
CREATE TEMPORARY TABLES, INDEX, ALTER
ON birdwatchers.*
TO 'admin_restore'@'localhost';
```

如此一来，Lena Stankoska 应该就可以恢复 `rookery` 和 `birdwatchers` 中的任何数据了。

13.3.3 用于批量导入的用户账号

我们需要给 Lena Stankoska 创建的最后一个管理员账号是 admin_import。她会使用该账号将数据从文本文件批量导入数据库。第 15 章将展开这个话题。这种数据导入方法需要使用 `LOAD DATA INFILE` 语句，而该语句仅需要 `FILE` 权限。

> `FILE` 权限存在安全风险，因为它可以读取服务器上 MySQL 能查看的任何文件。因此，只向用于导入文件的用户账号授予该权限，这一点尤为重要。而且，该账号的密码也应该只交给值得信任的人。除此之外，还可以通过 `secure_file_priv` 变量来限制只能读取某个目录。这样，文件系统的风险就降到最低了。甚至还可以在不执行导入操作时回收权限，而等到要导入时再授予。

`FILE` 权限不能指定用于某个库或某个组件，它是一个全局权限。一旦授予 admin_import@

localhost 这个权限，那么它就可以将数据导入任何数据库，并从任何数据库中导出数据，其中包括 mysql 数据库。所以，要留心这种权限的分配，并且绝对不要允许远程登录使用它。下面，我们来创建 admin_import@localhost，并给该账号授权：

```
CREATE USER 'admin_import'@'localhost'
IDENTIFIED BY 'another_pwd_789';

GRANT FILE ON *.*
TO 'admin_import'@'localhost';
```

现在，Lena Stankoska 的管理员账号都创建好了，并也分配好必要的权限（这些权限不多不少），供她操作数据库。现在，我们再来创建一个可能对你有用的管理员账号。

13.3.4 用于授权的用户账号

你可能还需要一个用于创建其他用户的用户账号。虽然可以直接用 root 来做，但为了贯彻按用途划分用户账号的思想，还是应该创建一个单独的用户账号，专门维护用户和权限。另外，这个任务应该交给我们不希望其完全控制数据库系统的人。

若要创建一个账号，使其能创建并授权其他用户账号，需要在 GRANT 中加上 GRANT OPTION 子句。它使这个账号能把自身拥有的权限授予其他账号——精度与原 GRANT 语句一样。如果原 GRANT 只有两个数据库，则给其他账号授权时也只能是这两个数据库。举个例子，执行以下命令，创建 admin_granter@localhost 这一账号，并以 GRANT OPTION 的方式给它授权：

```
GRANT ALL PRIVILEGES ON rookery.*
TO 'admin_granter'@'localhost'
IDENTIFIED BY 'avocet_123'
WITH GRANT OPTION;

GRANT ALL PRIVILEGES ON birdwatchers.*
TO 'admin_granter'@'localhost'
IDENTIFIED BY 'avocet_123'
WITH GRANT OPTION;
```

这样就创建了 admin_granter@localhost，它能把自己在 rookery 和 birdwatchers 上的权限授予别的用户账号。

不过，这样的权限对于管理其他账号的工作来说，仍然不够。例如，要是你希望这个用户账号能创建和删除账号，还得授予它 CREATE USER 权限。这样它可以执行 SHOW GRANTS，还需要 mysql 数据库上的 SELECT 权限。这又存在安全风险，要留意得到这个特权的用户账号。要授予用户账号这两个附加的特权，输入以下两条语句：

```
GRANT CREATE USER ON *.*
TO 'admin_granter'@'localhost';

GRANT SELECT ON mysql.*
TO 'admin_granter'@'localhost';
```

现在，admin_granter@localhost 可以进行用户账号管理的工作了。下面来测试一下，先

在命令行输入下面的第一条命令，登录 MySQL，然后在 mysql 客户端中输入剩下的 SQL 语句：

```
mysql --user admin_granter --password=avocet_123

SELECT CURRENT_USER() AS 'User Account';

+-------------------------+
| User Account            |
+-------------------------+
| admin_granter@localhost |
+-------------------------+

CREATE USER 'bird_tester'@'localhost';

GRANT SELECT ON birdwatchers.*
TO 'bird_tester'@'localhost';

SHOW GRANTS FOR 'bird_tester'@'localhost';

+-----------------------------------------------------------+
| Grants for bird_tester@localhost                          |
+-----------------------------------------------------------+
| GRANT USAGE ON *.* TO 'bird_tester'@'localhost'           |
| GRANT SELECT ON `birdwatchers`.* TO 'bird_tester'@'localhost' |
+-----------------------------------------------------------+

DROP USER 'bird_tester'@'localhost';
```

测试成功。整个过程如下：首先用 admin_granter@localhost 登录，并用 CURRENT_USER() 确认是否真的在用 admin_granter@localhost；然后，创建一个新账号，并授予它在 birdwatchers 数据库上的 SELECT 权限；接着，用 SHOW GRANTS 验证是否授权成功；最后，用 DROP USER 删掉该账号。我们可以把这个账号交给某个负责管理数据库账号的员工。

13.4 回收权限

从本章开头到现在，我们一直都在给用户账号授予权限。但是，有时候你可能想回收权限，也许是因为错误授权，或想改变用户账号的访问权限，抑或想改变所要保护的表。

REVOKE 可用于回收所有或部分已授予某个用户账号的权限。它有两种写法：一种用于回收所有权限，另一种用于回收指定的权限。下面就来看看这两种写法。

假设有个用户叫 Michael Stone，他要请几个月的假，并且其间不会访问我们的数据库。虽然可以删掉这个账号，但我们还是决定只回收权限，然后在他回来上班时再重新授权。具体来说，我们要用如下命令：

```
REVOKE ALL PRIVILEGES
ON rookery.*
FROM 'michael_stone'@'localhost';

REVOKE ALL PRIVILEGES
```

```
ON birdwatchers.*
FROM 'michael_stone'@'localhost';
```

这种写法与授予所有权限的 GRANT 相似。只是 GRANT 带 ON 子句，而 REVOKE 带 FROM 子句，以表示从哪个账号回收权限。尽管给 Michael Stone 授权时指定了某些表，但在回收时不需要逐个表回收，只需如上所写，便全部搞定。但在重新授权时，还是要逐个表来写 GRANT。

而第二种写法用于只回收部分权限。这些权限需要用逗号分隔，并写在 REVOKE 后面。权限名称与 GRANT 所用的一样（见表 13-1）。同样，可以逐个表写 REVOKE，也可以用星号来指代所有表。如果是回收某些列上的权限，就将列写在括号中，并以逗号分隔，就与 GRANT 一样。下面来看一个例子。

为了提高安全性，我们打算回收一些不必要的权限。之前我们为 admin_restore@localhost 授予了 ALTER 权限，但现在发现它从没被用过，所以，我们用如下语句回收该权限。

```
REVOKE ALTER
ON rookery.*
FROM 'admin_restore'@'localhost';

REVOKE ALTER
ON birdwatchers.*
FROM 'admin_restore'@'localhost';
```

13.5　删除用户账号

DROP USER 语句可用于删除用户账号。现在来看看它的用法。假设 Michael Stone 告诉我们他找到新工作了，不会再回来。那么我们可以用以下语句删除他的账号：

```
DROP USER 'michael_stone'@'localhost';
```

如果 MySQL 的版本较低（低于 5.0.2），则需要先回收所有权限，才能删除账号。先执行 REVOKE ALL ON *.* FROM 'user'@'host'，再执行 DROP USER 'user'@'host'。

有些用户可能拥有多个私人用户账号，例如 Lena Stankoska。因此，在删除 Michael Stone 的账号时，应该检查他是否还有其他账号。但可惜的是，MySQL 没有 SHOW USERS 语句。所以，我们只能检查 mysql 数据库中的 user 表：

```
SELECT User, Host
FROM mysql.user
WHERE User LIKE '%michael%'
OR User LIKE '%stone%';

+--------------------+-------------+
| User               | Host        |
+--------------------+-------------+
| mstone             | mstone_home |
| michael_zabbalaoui | localhost   |
+--------------------+-------------+
```

看来 Michael Stone 还有另一个与他家 IP 地址关联的账号。于是，跟他确认过之后，我们将该账号删除：

```
DROP USER 'mstone'@'mstone_home';
```

在删除账号时，如果该账号已登录，并且有活动中的会话，那么这些会话都不会被停止。它们会一直存活到用户退出或活动停止。不过，也可以即时结束会话。首先，执行以下命令，获取会话标识：

```
SHOW PROCESSLIST;
...

*************************** 4. row ***************************
      Id: 11482
    User: mstone
    Host: mstone_home
      db: NULL
 Command: Query
    Time: 78
   State: init
    Info: SELECT * FROM `birds`
Progress: 0.000
```

我将结果简化了。可以看到，虽然账号已被删除，但 mstone@mstone_home 仍有一个活动会话。因为该用户已经离职，并且没打算回来，所以我们担心他会从家里捞取数据。为了解决这个问题，我们可以用以下语句结束会话：

```
KILL 11482;
```

注意，我们使用了 `SHOW PROCESSLIST` 结果中的会话标识。`SHOW PROCESSLIST` 和 `KILL` 分别需要 `PROCESS` 和 `SUPER` 权限。因为现在这个会话和 Michael Stone 的用户账号都已被删除，所以他不可能再访问我们的数据库。想做得更严的话，我们甚至可以在操作系统层面删除他的账号，不过这个话题已超越了本书的范围。

13.6 更改密码和用户名

如果想使数据库更加安全，可以定期更改用户账号的密码（尤其是那些拥有管理权限的账号）。下一小节将讲解如何更改密码。而重命名账号，尽管不常见，但因为它可能会导致一些安全问题，所以也要引起注意。在改名或改密码时，应检查现存的脚本中是否用到它们，特别是要看看那些用来自动备份数据库的脚本。如果是，那么脚本也需要修改。

13.6.1 给用户账号设置密码

我们在本章创建用户账号时，有些带有密码，有些则没有。有时，你可能需要改密码（为了安全，确实应该定期改密码），可以用 `SET PASSWORD` 语句以及 `PASSWORD()` 函数（把密码加密）。

在 MySQL 5.6 版本中，可以令密码失效，以强迫用户修改密码。具体做法就是在 ALTER USER 中使用 PASSWORD EXPIRE 子句，如下：

```
ALTER USER 'admin_granter'@'localhost' PASSWORD EXPIRE;
```

这样，在用户下一次登录或执行 SQL 语句时，就会收到一个报错信息，告知需要改密码。用户必须改密码（使用 SET PASSWORD），不然无法执行任何 SQL 语句。

下面我们来修改 admin_granter@localhost 的密码：

```
SET PASSWORD FOR 'admin_granter'@'localhost' = PASSWORD('some_pwd_123');
```

这个密码太简单了，我们来改个复杂的密码，比如 P1ed_Avoce7-79873。若想更安全，可以在个人计算机上加密，然后再把加密后的密码填到服务器。假设个人计算机上有 MySQL，在命令行执行以下语句：

```
mysql -p --skip-column-names --silent \
      --execute="SELECT PASSWORD('P1ed_Avoce7-79873')"

*D47F09D44BA0456F55A2F14DBD22C04821BCC07B
```

返回的结果就是加密后的密码。只需复制它，然后登录服务器，再把它粘贴到改密码的命令中：

```
SET PASSWORD FOR 'admin_granter'@'localhost' =
'*D47F09D44BA0456F55A2F14DBD22C04821BCC07B';
```

密码会即时更新。你可以自己试试，看是否能用 P1ed_Avoce7-79873 登录。

如果忘了 root 密码，有一个简单的重置方法。首先，新建一个文本文件，输入以下内容，注意一行写一条语句：

```
UPDATE mysql.user SET Password=PASSWORD('new_pwd') WHERE User='root';
FLUSH PRIVILEGES;
```

将该文件起名为 rt-reset.sql，并放在受保护的目录中。然后用 --init-file，启动 MySQL：

```
mysqld_safe --init-file=/root/rt-reset.sql &
```

启动后，登录 MySQL，看看密码是否已经修改。可以使用这种方式多次修改密码。改完之后，删除 rt-reset.sql。还可以重启 MySQL，但因为不用再改了，所以可以不带 --init-file。

13.6.2 用户账号重命名

RENAME USER 用于修改用户名。可以用它来改用户账号的用户名和主机。只有具备 CREATE USER 权限和对 mysql 数据库 UPDATE 权限的账号，才能给其他账号改名。

为了看看 RENAME USER 的用法，我们就来试试把 lena_stankoska@lena_stankoska_home 改成 lena@stankoskahouse.com（假设 Lena Stankoska 是该域名的所有者，并会从该域名访问我

们的数据库）。输入以下命令：

```
RENAME USER 'lena_stankoska'@'lena_stankoska_home'
TO 'lena'@'stankoskahouse.com';
```

一旦执行以上命令，与 lena_stankoska@lena_stankoska_home 关联的所有权限都会移到 lena@stankoskahouse.com。下面来检查是否真的如此：

```
SHOW GRANTS FOR 'lena'@'stankoskahouse.com';

+-----------------------------------------------------------------------------+
| Grants for lena@stankoskahouse.com                                          |
+-----------------------------------------------------------------------------+
| GRANT USAGE ON *.* TO 'lena'@'...' IDENTIFIED BY PASSWORD '...'             |
| GRANT SELECT ON `rookery`.* TO 'lena'@'...'                                 |
| GRANT SELECT ON `birdwatchers`.`eastern_birders_spottings` TO 'lena'@'...'  |
| GRANT SELECT ON `birdwatchers`.`membership_prospects` TO 'lena'@'...'       |
| GRANT SELECT ON `birdwatchers`.`survey_answers` TO 'lena'@'...'             |
| GRANT SELECT ON `birdwatchers`.`surveys` TO 'lena'@'...'                    |
| GRANT SELECT ON `birdwatchers`.`survey_questions` TO 'lena'@'...'           |
| GRANT SELECT ON `birdwatchers`.`eastern_birders` TO 'lena'@'...'            |
| GRANT SELECT ON `birdwatchers`.`prospects` TO 'lena'@'...'                  |
| GRANT SELECT ON `birdwatchers`.`prize_winners` TO 'lena'@'...'              |
| GRANT SELECT ON `birdwatchers`.`possible_duplicate_email` TO 'lena'@'...'   |
| GRANT SELECT ON `birdwatchers`.`birdwatcher_prospects_import` TO 'lena'@'...'|
| GRANT SELECT (membership_type, human_id, name_last, formal_title, name_first)|
|       ON `birdwatchers`.`humans` TO 'lena'@'...'                            |
| GRANT SELECT ON `birdwatchers`.`bird_identification_tests` TO 'lena'@'...'  |
| GRANT SELECT ON `birdwatchers`.`birdwatcher_prospects` TO 'lena'@'...'      |
| GRANT SELECT ON `birdwatchers`.`bird_sightings` TO 'lena'@'...'             |
| GRANT SELECT ON `birdwatchers`.`birding_events` TO 'lena'@'...'             |
| GRANT SELECT ON `birdwatchers`.`random_numbers` TO 'lena'@'...'             |
+-----------------------------------------------------------------------------+
```

在 `Grants` 表中，你会发现该账号有很多权限。这是因为我们是按表和列来给它授权，而不仅仅是数据库层面的授权。但更重要的是，这些权限现在都属于 lena@stankoskahouse.com 了。

13.7 用户角色

给一个人创建多个用户账号是很繁琐的工作。试想在你的管理范围内，有很多像 Lena Stankoska 这样的用户，你需要给他们每人创建多个账号。如果有位用户因为要顶替某个休假的人，而需要在一段时间内接管一些权限，那你就得为他授予额外的权限，然后过段时间又将它们回收。这是很繁琐的，会引发安全问题（例如，不小心授予了太多权限），而且令控管变得低效（检查这么多账号很困难）。所以，我们要用更好的方法。

这个方法就是在 MariaDB 10.0.5 版本中引入的*用户角色*，而 MySQL 是没有的。用户角色让你可以创建一个更高层次的概念，即*角色*，并将它授予指定的用户账号。一般来说，每个用户账号都有能完成日常任务的所需权限，而当需要做一些非日常的工作时，可以临时赋予它某种特定的角色，等它完成相关工作之后，再取消这次“赋予”。这样做很方便。

下面来看一个示例。

早前，我们给 Lena Stankoska 创建了一个具有 FILE 权限的账号 admin_import，使她能执行 LOAD DATA INFILE 语句，把文本文件中的数据导入数据库（此语句及其过程将在第 15 章讲解）。现假设还有另外两个用户，Max Mether 和 Ulf Sandberg，他们也会偶尔做这项工作。虽然我们可以不用再给他们分别创建用于导入数据的账号，而只给他们 admin_import 的密码，但这样做不够专业、不够安全。比较好的做法是，用 CREATE ROLE 语句来创建一个名为 admin_import_role 的角色，并把该角色赋予 Max Mether 和 Ulf Sandberg。

如果你安装的是 MariaDB，那么可以输入以下命令：

```
CREATE ROLE 'admin_import_role';

GRANT FILE ON *.*
TO 'admin_import_role'@localhost;
```

第一条语句用于创建角色。而第二条语句用 GRANT 来给该角色授予导入文件所需的 FILE 权限。接下来，让我们给 Max Mether 和 Ulf Sandberg 赋予这个角色（假设这两个人的用户账号都创建好了）。在 MariaDB 中，输入以下命令：

```
GRANT 'admin_import_role' TO 'max'@localhost;
GRANT 'admin_import_role' TO 'ulf'@localhost;
```

这样一来，Max Mether 和 Ulf Sandberg 便可以在有需要时，担任这个角色。比如说，Max Mether 可以在登录 MariaDB 后，执行以下命令：

```
SET ROLE 'admin_import_role';

LOAD DATA INFILE
...

SET ROLE NONE;
```

如你所见，Max Mether 将自己设为 admin_import_role 后，执行了 LOAD DATA INFILE 语句。这里，我隐藏了该语句的细节以及他可能输入的其他语句，以便我们只关注用户角色这一话题。最后，他将自己的角色设为 NONE，以脱离角色。

> 角色有一个缺点：它只在当前会话有效。当使用其他外部实用程序（如 mysqldump）时，这就有点麻烦。如果在命令行开启 mysql 客户端，并为用户账号设置角色，然后退出 mysql 或在另一终端执行 mysqldump，那么 dump 会在另一个会话中。而在该会话中没有设置角色，所以没有权限。

用户角色很好用，而且比创建一大堆用户账号和密码并给每一个授权要简单得多。它是临时授权的理想工具。对于管理员来说，它能使用户账号和权限的管理更轻松。而用户则只需输入一个用户名和密码，即可进行所有的工作。必要的时候，他们只需要被赋予一个角色。当然，应该只在必要时才设置角色，并在完事后设回 NONE。

13.8 小结

在你刚开始做数据库管理员的时候，可能只想创建少量的用户账号，甚至只想使用 root。但其实，应该学会使用各种用户账号，而不是只用 root。另外，也应该学会给每个人分配至少一个私人账号（如果可以的话，尽量不允许分享账号）。还要学会给每个账号仅授予必要的权限。这种做法可能很乏味，却很安全——它除了能保护敏感数据，还能防止数据丢失以及表被误改或误删。

其实我们还未谈及一些关于用户账号和安全措施的选项。例如，可以用一些选项限制某个账号每次或每小时的连接数。而密码的加密和解密函数也还有不少。因为考虑到你不会经常使用它们（特别是对于新手来说），所以本书不讲这些。可以在我的另一本书《MySQL 核心技术手册》中（http://shop.oreilly.com/product/9780596514334.do），或 MySQL 资源网上（http://mysqlresources.com/）学习这些内容。

13.9 习题

尽管你可以随时回到本章翻看 `CREATE USER`、`GRANT`、`REVOKE` 和 `DROP USER` 的用法，但应该掌握好它们，不要每次用时都翻书。要想记住它们的语法，可以借助 `SHOW GRANTS`。如果你熟悉它们，那么会更愿意调整用户账号的权限。否则，只好让数据库部门的员工分享同一个账号，并给该账号分配所有权限。为了让你对它们有更深刻的印象，我给你准备了以下习题。不过说到底，能否贯彻安全措施，还得看个人意志。

(1) 登录你的服务器，使用 `CREATE USER` 创建一个管理员账号，用户名为 admin_boss，主机名为 localhost。
   然后，使用 `GRANT`，让这个账号在 `rookery` 和 `birdwatchers` 上拥有 `ALL` 权限，并授予它 `SUPER`，使它能更改服务器设置。同时，带上 `GRANT OPTION` 权限（13.3.4 节讲过）。你可能需要分开几次写 `GRANT`。记住，至少使用一次 `IDENTIFIED BY` 子句，这样才能给它设置密码。
   创建好这个账号之后，试试退出 MySQL，并用 admin_boss 重新登录，看密码是否有效。之后，不再用 root，而改用它。

(2) 以 admin_boss 登录之后，用 `GRANT` 语句创建一个从 localhost 登录且名为 sakari 的用户。只让它在 `rookery` 和 `birdwatchers` 上有 `SELECT`、`INSERT` 和 `UPDATE` 权限。记得给它设置密码。要求所有授权都写在一个 `GRANT` 中。做完后，退出 MySQL。
   用刚才创建的 sakari@localhost 登录 MySQL。执行 `SHOW DATABASES`，确认是否只能看到两个默认数据库和两个我们刚才授权的数据库。然后用 `SELECT` 语句查询 `birdwatchers` 数据库中 `humans` 表的所有行，接着用 `INSERT` 插入少量数据，再用 `UPDATE` 修改至少一列你刚才录入的数据。正常来说，你应该有权限完成这些操作。如果不能，用 admin_boss 登录，再用 `SHOW GRANTS` 检查 sakari@localhost 的权限设置。改正错误，然后再测试这个用户账号。
   接着，尝试用 `DELETE` 删除刚才加入的行（用 sakari@localhost 登录，而非 admin_boss）。如无意外，应该是不能执行的。

(3) 以 admin_boss 登录后，用 `REVOKE` 回收在第 2 题中授予 sakari@localhost 的 `INSERT` 和 `UPDATE` 权限。做完后，退出 MySQL。

以 sakari@localhost 登录，尝试用 `INSERT` 往 `humans` 表插入一行数据。不过这应该是不成功的。如果能插入，则以 admin_boss 登录，想想刚才回收权限时有什么搞错了，并修正。然后再试试用 sakari@localhost 插入数据。

(4) 以 admin_boss 登录，修改 sakari@localhost 的密码（13.6 节讲过）。做完后，退出 MySQL。

使用新密码以 sakari 登录，然后按几次向上的方向键，看看有没有出现 sakari@localhost 的密码。如果有，那就意味着其他用户有可能也会看到这个密码。检查过后，退出 MySQL。

在自己的计算机（不是服务器）的命令行中，用 `mysql` 执行 `SELECT` 语句以及 `PASSWORD()` 函数，生成一个经过加密的密码，以用于 sakari@localhost。记得要与之前的密码不同。若忘了怎么做，可参考 13.6 节。

然后，以 admin_boss 登录，把 sakari@localhost 的密码设置成刚才加密后的密码，这次不需要用 `PASSWORD()`。然后退出并用新密码以 sakari 登录。用向上的方向键确认密码是加密后的字符串，而非明文。

(5) 以 admin_boss 登录，用 `DROP USER` 删除 sakari@localhost。然后退出，再试试能不能以 sakari 登录。正常来说，是不能的。

第 14 章 数据库的备份与恢复

数据库的创建通常要依靠许多人的力量，有时人数甚至高达数千。创建数据库的机构聘请开发和管理人员，除此之外还有提供数据的人，他们可能是雇员，也可能是会员。但数据库的大部分数据可能来自其他人，例如客户以及通过网络提供数据的无名人士。数据量可能非常大。即使是一个小网站，它也可能会积累数千行数据。而大网站更可能有数百万行数据。数据的收集需要大量的人力，而数据很容易丢失，一个小小的硬盘错误就会导致悲剧发生。所以，我们需要以正确的方式定期进行数据备份：数据非常多，而且很多任务都依赖它。

如果你准备做数据库管理员，那一定要懂得如何备份和恢复数据库。需要计划好哪些数据要备份，什么时候做备份，以及备份到哪里。此外，还要偶尔检查是否备份成功，以防要做恢复时才发现没备份好。还应熟习如何恢复备份，以便出现问题时能快速解决。以上提到的这几点，本章都会介绍。

14.1 备份

`mysqldump` 是 MySQL 和 MariaDB 上最好的备份工具之一。它包含两个服务器，而且不需要任何成本。也许你已经安装它了。它最大的优点是，无需关闭服务器就能做备份（如果对数据一致性有很高要求，那你可能需要规范它的使用）。备份工具还有很多（比如 MySQL Enterprise Backup 和 Percona XtraBackup），其中有些带了 GUI，有些功能更多。你可以从 Sveta Smirnova 写的《MySQL 排错指南》中了解其他类型的备份和工具。不过，`mysqldump` 是最流行的备份工具，所以，作为管理员新手，你应该熟悉使用它的方法（就算你以后会用到商业版工具）。而本章也将用它来演示。

它其实很简单：首先查询每个数据库和每个表的结构与数据，然后把查出的所有内容导出到文本文件中。它创建的默认文本文件被称为 dump 文件，里面包含重建数据库和数

据必需的 SQL 语句。如果打开这个文件，你会看到一些 CREATE TABLE 语句，以及大量的 INSERT 语句。这看似累赘，但实际上很简单，且便于管理。

mysqldump 提供很多选项。可以备份所有数据库，也可以指定只备份某些数据库，甚至可以指定只备份某些表。本节就来介绍这些选项，并给出一些例子，告诉你它们的常用组合。

14.1.1 备份所有数据库

备份所有数据库、所有表，以及其中的数据，是最简单的。用 mysqldump 很容易就能做到。现在就请在服务器的命令行中，使用上一章创建的 admin_backup 账号，执行以下命令。你可能需要将 /data/backups/ 改成你的服务器上存在的路径。又或者，不写目录路径，而在当前目录下创建 dump 文件。

```
mysqldump --user=admin_backup \
          --password --lock-all-tables
          --all-databases > /data/backups/all-dbs.sql
```

这个命令用到了以下选项。

- --user=*admin_backup*

  让 mysqldump 以 admin_backup 账号与 MySQL 服务器进行交互。13.2.1 节已演示过如何创建这一账号，所以，如果你当时没有跟着做，那现在就得创建一个拥有适当权限的用户账号。虽然你可能想用 root 来做备份，但另建一个管理员账号来处理这种事情，才是正确的做法（就像这里用 admin_backup）。用于备份的账号只需要锁表和读表的权限。

- --password

  让 mysqldump 在下一行弹出输入密码的提示符。它的效果如同 mysql。如果这个备份命令要写在让 cron 执行的 shell 脚本中，那这个选项就要写成 --password=my_pwd（请在 my_pwd 处填真实密码，即明文写出）。所以，不要用 root，以免暴露密码。

- --lock-all-tables

  在做备份前，先让 MySQL 锁住所有表，然后直到备份完成才解锁。对于繁忙的数据库来说，长时间锁住所有表会有很大影响。后面我们会介绍其他锁定方法。

- --all-databases

  导出所有数据库。下一小节会介绍另一个能指定备份部分数据库的选项。

命令行中的大于号是让 shell 把标准输出（STDOUT）重定向，使备份能被导入后面指定的文件。请按你的系统和喜好来设置该文件的路径和名字。

这样得到的 dump 文件就会是一行数据一条 INSERT。如果想一个表一条 INSERT，可加上 --extended-insert 选项，如此一来，生成的 dump 文件会小一些，而且在恢复数据库时，比一行数据一条 INSERT 更快。如果你的服务器默认在 dump 文件中生成扩展插入，而你更喜欢它们作为单独的语句，那就用 --skip-extended-insert 选项。

生成的 INSERT 语句不含列名，它只是把值按列的顺序来排。如果想加上列名，那就加上 --complete-insert 选项。

在 mysqldump 后带选项时，选项的排列顺序无关紧要，但其取值必须紧跟选项。此外，重定向符号必须写在最后。准确来说，它不是 mysqldump 的选项，而是 shell 运算符。总的来说，选项的排列顺序类似于命令的排列顺序。

MySQL 的工具过去常常提供一些简短形式的选项，比如用 -u 代表 --user。但是这些简短形式现在已不被推荐，甚至在未来的版本中可能会被取消。

用 mysqldump 给 InnoDB 表或其他事务型的表做备份时，最好加上 --single-transaction 选项。这能提高数据的一致性。直到 dump 完成，它才会在表中做改动。但是，这个选项会令 --lock-tables 失效，也就是说不能保证同一个数据库中 MyISAM 表的数据一致。要避免这种问题，可以对整个数据库的表都用同一种存储引擎，又或者分别备份 InnoDB 表和 MyISAM 表。

用 mysqldump 一次性备份所有数据库，可能会令 dump 文件过大。对于定期备份的小型数据库来说，这是可以接受的。但对于大型数据库来说，完成备份就会很花时间，并且，在这个过程中一直锁表会很妨碍别人的操作，最后恢复起来也很麻烦。如果不想出现这种问题，就需要采取一些巧妙的方法。例如，分开备份每个大数据库，这样就能将原本较大的 dump 文件分成几个小一些的 dump 文件。也可以选择在较为空闲的时间备份那些平时很活跃的大数据库。关于如何分库备份以及如何决定备份策略，我们会在以后讨论。现在，先来熟悉 dump 文件。

备份所有数据库会有一个安全问题，因为这将把 mysql 数据库的 user 表也导出来。而该表含有用户名和密码。如果想把它排除在外，可以在创建 dump 文件时，给 mysqldump 命令加上 --ignore-table=mysql.user。而如果只想偶尔备份 mysql.user，最好另开一个账号，并且指定导出到受保护的目录或其他安全的地方。

14.1.2　理解dump文件

运行完上一小节的 mysqldump 后，用普通的文本编辑器打开生成的 dump 文件，看看它的内容。你会发现它是这样的：mysqldump 先给 dump 文件写注释，设置一些变量，然后列出 CREATE DATABASE、CREATE TABLE 和许多 INSERT 语句。下面我们就来研究一下部分内容，以更好地理解 dump 文件。这对恢复数据库也是有帮助的。

首先看看它的头部。以下是上例所产生的 dump 文件的前几行：

```
-- MySQL dump 10.14 Distrib 5.5.39-MariaDB, for Linux (i686)
--
-- Host: localhost    Database: rookery
-- ------------------------------------------------------
-- Server version      5.5.39-MariaDB
```

第一行显示的是 mysqldump、MySQL（本例中是 MariaDB）和操作系统的版本。接着我们看到，这次 dump 是从 localhost 登录后执行的。并且，在同一行，文件标明了首先备份的

数据库的名字。然后，在分割线下方，是 MariaDB 的版本。虽然第一行已有版本信息，但这里单独写出，显得更清楚。

接着就是一大堆 `SET` 语句，如示例 14-1 所示。

示例 14-1：dump 文件中的条件性 SET 语句

```
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;
```

这些 `SET` 语句处于 `/*` 和 `*/` 之间，看起来像是不会被执行的注释。但看清楚，开头其实是 `/*!`，而不只是 `/*`，这是 MySQL 和 MariaDB 的条件性语句。在做恢复时，MySQL 和 MariaDB 会检查感叹号后的版本号是否与自身匹配，再决定是否执行。而在 dump 文件中，注释以 `--` 开头。

可以在 `mysqldump` 后加上以下一个或多个选项，以减小 dump 文件的大小。

- `--skip-add-drop-table`

  不带 `DROP TABLE` 语句，即不在恢复时删除原表。

- `--skip-add-locks`

  不带锁表语句。

- `--skip-comments`

  不带注释。

- `--skip-disable-keys`

  不带使表中的索引暂时失效的语句。

- `--skip-set-charset`

  不带 `SET NAMES`，即不在恢复时设置字符集。

- `--compact`

  使用上述所有选项。

上述选项中的某些是有风险的。例如，做恢复时所在的数据库的默认字符集可能与原数据库的不同，如果使用 `--skip-set-charset`，那么就可能用错字符集。而如果使用 `--skip-add-locks`，则可能导致数据不一致。

因为 dump 文件不仅能用于在原服务器上备份和恢复数据，还可以用于将数据复制到另一个服务器，所以需要用条件性语句来判断在另一个服务器上应执行什么命令。这样才能使创建表和插入数据时不会出错。执行 dump 文件时，它会让数据库和表得以恢复或重建成原本的样子。

现在我们回顾第一个 SET 命令：

```
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
```

它一开始就指定只在 MySQL 或 MariaDB 4.01.01 及以上版本执行此命令。mysqldump 就是用这种方式来确保新版才有的功能不会在旧版中被调用。我们通常认为，一旦某个版本支持一个功能，那么未来的版本都会继续支持它。而接着的语句，就是另存全局变量 CHARACTER_SET_CLIENT 当前的值。如果回顾示例 14-1，会发现在后续的命令中，CHARACTER_SET_RESULTS 和 COLLATION_CONNECTION 也被另存了。然后，第四行将这三个变量都设成 utf8，因为 NAMES 指代这三个变量。

再跳到 dump 文件的末尾，会看到如下一堆 SET 语句：

```
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2014-09-14  6:13:40
```

但是，它们与刚才的顺序相反。这些 SET 语句所做的就是用刚开始创建的变量将这些全局变量恢复成原本的设置。在 dump 文件中，你会看到很多这样的条件性语句。而为了让用户的操作不受这些被改变的全局变量影响，应该在做恢复时进行锁表。

现在回到开头的条件性语句，看看其后有什么：

```
--
-- Current Database: `rookery`
--

CREATE DATABASE /*!32312 IF NOT EXISTS*/ `rookery`

/*!40100 DEFAULT CHARACTER SET latin1 COLLATE latin1_bin */;

USE `rookery`;
```

头三行注释提示你，当查看 dump 文件时，你会知道这部分内容与 rookery 数据库有关。而第一条语句理所当然是 CREATE DATABASE。不过，它可能有点迷惑人，因为它包含了两条按版本来执行的条件性语句。下面我们就来看看其中一条语句。

在这条 SQL 语句中，如果 MySQL 的版本是 3.23.12 及以上，则会执行 IF NOT EXISTS。这个版本够旧，而 IF NOT EXISTS 就是在这版引入的，而且一直存在至今。虽然现在应该不会有服务器还用着这么旧的 MySQL，但 mysqldump 还是会加上这个检查，以防万一。而 IF NOT EXISTS 本身则更值得关注。它的作用是检查 rookery 数据库是否存在，如果存在，则不用 CREATE DATABASE 来创建该数据库。顺便说一下，如果不想让 dump 文件带有 CREATE DATABASE 和 CREATE TABLE，可以在 mysqldump 后加上 --no-create-info 选项。

最后一条 SQL 语句将默认数据库切换为 rookery。你可能会觉得奇怪，为什么 mysqldump 要用 USE 来指定数据库，而不是在后面的 SQL 语句中加上数据库的名字（即不写成 INSERT INTO `rookery`.`bird_families`...）？虽然这好像无关紧要，但其实用 USE 是有好处的。当你想用 dump 文件中的表和数据在同一个服务器上创建新的数据库时，只需编辑 USE 语句，在一个地方改动数据库的名字即可（例如，将 rookery 改成 rookery_backup）。这样，就创建了一个与原数据库一模一样的副本，而原数据库也得以保留。我们会在后面深入讲解这个问题。现在先看看 dump 文件中接下来还有些什么。

下一部分是处理 rookery 的第一个表。如下所示，我们看到了 bird_families 的表结构：

```
--
-- Table structure for table `bird_families`
--

DROP TABLE IF EXISTS `bird_families`;

/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;

CREATE TABLE `bird_families` (
  `family_id` int(11) NOT NULL AUTO_INCREMENT,
  `scientific_name` varchar(100) COLLATE latin1_bin DEFAULT NULL,
  `brief_description` varchar(255) COLLATE latin1_bin DEFAULT NULL,
  `order_id` int(11) DEFAULT NULL,
  PRIMARY KEY (`family_id`),
  UNIQUE KEY `scientific_name` (`scientific_name`)
) ENGINE=MyISAM AUTO_INCREMENT=334 DEFAULT CHARSET=latin1 COLLATE=latin1_bin;

/*!40101 SET character_set_client = @saved_cs_client */;
```

这里的第一句可能会令你警惕。这是应该的，因为这条 DROP TABLE 语句正要删除 bird_families 表。按理说，即使我们删除该表，也不应该会丢失数据，因为紧接着我们就会重建该表，并把创建 dump 文件时该表拥有的数据都导回去。但是，如果在 dump 文件创建后，bird_families 表中的数据有改动的话，那么当我们把表恢复到之前的状态时，这些改动就会丢失。如果想加上这些改动，需要采取一些方法。一种方法是，像之前提到的，修改 USE 语句，将所有的结构和数据导入一个临时数据库，然后就可以在两个数据库之间进行一些合并。在某些情况下，可以考虑使用 REPLACE 而不是 INSERT 来进行合并。另一种方法，就是去掉 DROP TABLE，并把 CREATE TABLE 中的表名换成新的。我们会在 14.2 节中讲解这些做法。

IF EXISTS 选项可以确保只有当表存在时，做恢复才会删表。如果表不存在，而又不加此语句，导致执行了 DROP TABLE，那么就会报错，使恢复中止。

DROP TABLE 之后是有关表和客户端配置的条件性语句，然后才是 CREATE TABLE。这个 CREATE TABLE 与 SHOW CREATE TABLE 看到的结果是一样的。最后，我们将变量重置回原状态。

这样，bird_families 就准备好导入数据了。所以，下一部分是如下语句：

```
--
-- Dumping data for table `bird_families`
--

LOCK TABLES `bird_families` WRITE;

/*!40000 ALTER TABLE `bird_families` DISABLE KEYS */;

INSERT INTO `bird_families` VALUES

...

/*!40000 ALTER TABLE `bird_families` ENABLE KEYS */;

UNLOCK TABLES;
```

注释之后有一条锁定 `bird_families` 的 `LOCK TABLES` 语句。它带有 `WRITE` 选项，使得恢复过程中其他人不能读取或修改该表的数据。这里会出现一个问题：`mysqldump` 备份哪个表，就只锁定该表的 `WRITE`，而让其他未备份的表依然可以被别人读写。你可能觉得这样挺好，但其实可能会造成数据不一致。

例如，假设在备份的过程中，你刚导完 `birdwatchers` 数据库的 `bird_sightings` 表的数据，正准备处理 `humans` 表。而就在这时，你执行了 `DELETE`，删掉了 `humans` 表中的某个人，以及 `bird_sightings` 中的相关条目。然后，`mysqldump` 开始备份 `humans` 表。那么，在恢复整个 `birdwatchers` 数据库时，你就会发现 `bird_sightings` 表中有某个人的记录，但此人在 `humans` 表中不存在。

如果你的数据库并不是经常有人访问，那么就不太会出现这个问题。但要是你想确保数据一致性，可以在执行 `mysqldump` 时加上 `--lock-tables` 选项。这样就能在备份之前，先把数据库里的所有表锁住，并在备份结束后才解锁。在备份多个数据库时，它也只会锁定正在备份的那个数据库，然后在备份下一个数据库前，将刚才的那个数据库解锁。如果你要保持多个数据库之间的一致性（换句话说，有一个数据库的数据依赖于另一个数据库），那么可用 `--lock-all-tables` 将所有数据库的所有表都锁定，直至整个备份完成。

在刚才的示例中，`LOCK TABLES` 后面还跟了条件性语句 `ALTER TABLE...DISABLE KEYS`，它的作用是使 `bird_families` 表的键失效。这种做法可令恢复的时间缩短，因为这样 MySQL 就不会在每次执行 `INSERT`（在示例的省略号处）时给所插入的条目建立索引，而在有关该表的所有 `INSERT` 完成后才建，即示例中的第二个 `ALTER TABLE`。`ENABLE KEYS` 所使用的算法很高效，适用于一次性对整个表迅速建立索引。

> 如果服务器默认使用 `--disable-keys`，那么 dump 文件就会带有 `DISABLE KEYS` 之类的条件性组件。若在 `mysqldump` 创建的 dump 文件中找不到这些组件，那就说明服务器并没有默认使用 `--disable-keys`。要解决这个问题，可以在执行 `mysqldump` 时加上它，或者将它加到 MySQL 配置文件的 `[mysqldump]` 部分中。

最后一句 `UNLOCK TABLES` 是对本部分开头锁定的表进行解锁。

总的来说，每个表的基本模式是：先建立表结构，再导入数据。为了建立表结构，dump 文件一般包含一些 SQL 语句，用于删表、设置临时变量、重建该表，以及恢复变量。为了在重建表时处理数据，它会先锁表，然后禁掉表上的键，再插入所有数据，最后重新启用键，并把表解锁。为数据库中的每个表都执行一遍这个过程，并且做完一个数据库，再做另一个，直到完成所有数据库。因为这个例子是备份所有数据库，所以你会在 dump 文件中看到这样的顺序。

dump 文件的内容会因 `mysqldump` 版本和服务器默认设置的不同而异。此外，它还会受备份命令所带的选项和各个数据库本身的特征影响。不过，看完以上例子，你应该已掌握如何阅读 dump 文件。下面，我们就回到用 `mysqldump` 执行备份这个话题上。

14.1.3　备份指定的数据库

在讲解 dump 文件的内容之前，我们已学过备份所有数据库的方法，但有些时候，你可能只想备份一个数据库或一些指定的数据库。那么到底要怎么做呢？

在这种情况下，就不应该用 `--all-databases` 了，而应该用 `--databases`，并在其后带上想备份的那些数据库的名字。例如，用以下命令来备份 `rookery`：

```
mysqldump --user=admin_backup --password --lock-tables \
          --verbose --databases rookery > rookery.sql
```

总的来说，与之前的写法差不多，只是这次我们指定了只备份 `rookery`。如之前所述，对于繁忙的服务器而言，分别备份多个数据库有利于减轻服务器的负担，并使恢复过程更容易掌控。顺便说一下，如果只想备份某个数据库的结构，而不需要备份数据，那么可以再带上 `--no-data`。这样出来的 dump 文件便只会有数据库和表的结构，而没有任何数据。

你可能已留意到，这里还用了 `--verbose` 选项。它会使 `mysqldump` 显示出备份过程中的每个主要步骤。在我们的环境中执行该命令，会出现如下结果：

```
-- Connecting to localhost...
-- Retrieving table structure for table bird_families...
-- Sending SELECT query...
-- Retrieving rows...
-- Retrieving table structure for table bird_images...
...
-- Disconnecting from localhost...
```

这些信息有时候会很有用，尤其是在出错时，它们能让你知道哪些表备份成功了，以及什么时候出现了问题。

如果是备份多个数据库，则在 `--databases` 后，将它们的名字以空格分隔列出（可能你会以为是用逗号分隔，但确实不是）。如下语句将备份 `rookery` 和 `birdwatchers` 数据库：

```
mysqldump --user=admin_backup --password --lock-tables \
          --databases rookery birdwatchers > rookery-birdwatchers.sql
```

此命令会将 `rookery` 和 `birdwatchers` 备份到 rookery-birdwatchers.sql 这个文件中。因为这两个数据库相互关联，而又与其他数据库无关，所以这是不错的做法。我们可以将此命令放到 `crontab` 或其他调度工具中，使这两个数据库能每天都自动备份。不过，这样的话，

每天的 dump 文件都会被覆盖一次。如果我们误删了数据，并在几天后才发现，那就不能从备份中恢复数据了。考虑到这种情况，我们需要每天都用一个不同的文件名来创建备份。而如果不想每次都亲自起名字，可以用 shell 脚本来实现。

14.1.4　创建备份脚本

若想自动化备份，可以编写脚本，使 `mysqldump` 按自己的设置被执行。这并不复杂。如果你的要求很简单，例如只是让每次备份的结果略微不同，那不需要有多高的编程技巧也能做到。

我们用上一小节末尾的那个问题，作为备份脚本的例子。要解决该问题，可以在 dump 文件的名称中加上日期，这样便可以每天都有一份独立的 dump 文件了。具体做法，可使用以下这个简单的 shell 脚本，它在 Linux 和 Mac 系统上都能运行：

```
#!/bin/sh

my_user='admin_back'
my_pwd='my_silly_password'

db1='rookery'

db2='birdwatchers'

date_today=$(date +%Y-%m-%d)

backup_dir='/data/backup/'
dump_file=$db1-$db2-$date_today'.sql'

/usr/bin/mysqldump --user=$my_usr --password=$my_pwd --lock-tables \
                   --databases $db1 $db2 > $backup_dir$dump_file

exit
```

此脚本在执行 `mysqldump` 时所带的选项，与之前的例子一样。首先，设定变量的用户名、密码和数据库名。然后，用 `date` 命令获取年月日的数字值（以横杠连接），存到变量 `date_today` 中。dump 文件的名字则是用数据库名（`$db1`、`$db2`）和 `$date_today` 拼接而成（结果大概会是 rookery-birdwatchers-2014-10-25.sql 这样）。最后，用所有这些变量拼接出我们要执行的完整的 `mysqldump`。

因为用户名和密码都写在脚本里，所以可以用 `cron` 来让它每天自动地执行，而无需再手动发起。它可以每天都创建一个新的 dump 文件，绝不覆盖旧的 dump 文件。但是，它仍有一些不足。例如，它不具备容错能力。当备份失败时，它不会通知任何人。另外，它也不会检查备份是否积压太多。一般来说，好的备份脚本应该能做到定期清理旧备份。当然，让脚本自动清理文件会有些麻烦。这里的例子只为给你提供创建备份脚本的基本思路。你所要创建和使用的备份脚本应该更加复杂，并需要处理错误或产生报表等各种功能。

14.1.5　备份指定的表

对于那些繁忙的大型数据库来说，你可能不想一次就备份整个数据库，而想逐个备份表。

你可能每周备份一次整个数据库，然后每日备份某些经常有数据改动的表。对于大部分数据库来说，都应该采取这种谨慎的做法。

以我们的两个数据库为例。rookery 数据库的数据不常更新：我们不可能天天都会发现新的鸟种，同理，鸟科和鸟目也是如此。一旦每种鸟的所有详细信息都确定下来，它们就基本上不会改变。而 birdwatchers 数据库则不是这样。如果我们的网站很繁忙，那么该数据库的所有表就会经常有新增和修改，所以，我们希望每天都备份一次。一个合理的做法就是，每周备份一次 rookery，每天备份一次 birdwatchers。

再假设我们的老板十分担心丢失会员录入的数据。他要求我们每天都做两次 humans 表的备份（中午和深夜各一次）。我们当然可以像上一小节那样写个脚本，在生成文件名时，加上 midday（中午）和 midnight（深夜），比如 birdwatchers-humans-2014-09-14-midday.sql 和 birdwatchers-humans-2014-09-14-midnight.sql。而唯一需要改变的是，让 mysqldump 只备份 humans 一个表。试着执行下面的命令：

```
mysqldump --user=admin_backup --password --lock-tables \
          --databases birdwatchers --tables humans > birdwatchers-humans.sql
```

与之前不同的是，这里加了 --tables 选项，并带上了表名。如果想备份同一个数据库中的多个表，只需在 --tables 后，以空格分隔的方式，列出这些表即可。其实以上命令不用这么长。因为只备份一个数据库中的表，所以 --databases 是不需要的。另外，因为 mysqldump 能把数据库名后的非保留字都当成表名，所以 --tables 也是不需要的。这样，上例的命令可以写成这样：

```
mysqldump --user=admin_backup --password --lock-tables \
          birdwatchers humans > birdwatchers-humans.sql
```

虽然这样写更简短，但之前那种写法更易于看出哪个是数据库名，而哪个是表名。

现在，尝试增加一个表，不过是另一个数据库中的表。假设老板说 rookery 中的 birds 表也要备份。不过，mysqldump 不能在有 --tables 的情况下带两个数据库，我们只能为两个数据库分别执行一次 mysqldump，而这将会生成两个 dump 文件。如果想让一个 dump 文件包含两个表，可以这样做：

```
mysqldump --user=admin_backup --password --lock-tables \
          --databases rookery --tables birds > birds-humans.sql

mysqldump --user=admin_backup --password --lock-tables \
          --databases birdwatchers --tables humans >> birds-humans.sql
```

此处我们运行了两次 mysqldump，但第二次用的是重定向追加符号（>>），令输出的内容位于同一个 dump 文件的末尾，而不是创建一个新的 dump 文件。这样，dump 文件的中间就会有一个注释，说这一次 dump 结束了，接着另一个注释说第二次 dump 开始。不过因为它们都是注释，所以是无关紧要的。另外，因为每次 dump 的变量设置都会在结束时恢复，所以前面的 dump 不会影响后面的 dump。简单地说，这样追加 dump 是没有问题的。

mysqldump 是易用而又强大的工具。虽然我们已经介绍了它的很多选项，但其实它还有更多可以继续挖掘的选项。详情可以参考 MySQL 和 MariaDB 的网站，或者我的另一本书《MySQL 核心技术手册》（http://shop.oreilly.com/product/9780596514334.do)。

而在用 dump 文件做恢复时需要很小心，否则有可能会破坏数据库。因此，应该多在测试数据库或测试服务器上练习恢复 dump 文件，以使自己对备份和恢复得心应手。不然，等到丢失数据时，你只能磕磕碰碰地去做恢复，这样可能会造成无法挽回的错误，或那时才发现原来数据都没有备份好。所以，还是准备好练习环境，踏实地掌握这些技能吧。下一节就来讲讲如何用备份文件恢复数据。

14.2 恢复备份

如果你的 MySQL 丢了数据，而你一直在用 `mysqldump` 定期备份数据，那么就可以通过这些 dump 文件来做恢复。毕竟，备份文件的目的正在于此。只需用 `mysql` 客户端执行 dump 文件中的所有语句，就可以恢复用 `mysqldump` 备份的 dump 文件。可以恢复所有数据库、单个数据库、某些表，甚至某些行。我们会在本节讲解这些内容。

14.2.1 恢复数据库

先来看看如何恢复整个数据库。为了安全起见，作为实验过程的一部分，我们给 `rookery` 做一次新的备份，然后恢复它。执行以下命令：

```
mysqldump --user=admin_backup --password --lock-tables \
          --databases rookery > rookery.sql
```

恢复前，先检查 dump 文件是否有该有的语句。如果正常，那就删掉 `rookery` 数据库。虽然这听起来很可怕，但不用担心，我们已做好备份。数据库崩溃或被意外删除是有可能发生的。所以，我们应提前在 `rookery` 这种测试数据库上演习一下，以建立信心。下面就来删除这个数据库：

```
mysql --user=admin_restore --password --execute "DROP DATABASE rookery;"
```

这里的做法是，在命令行中用 `mysql` 执行 `DROP DATABASE` 语句（当然，登录 `mysql` 客户端再执行 `DROP DATABASE` 也可以），这需要用到 `--execute` 选项。另外，你也需要指定一个有删除数据库权限的管理员。而这里就用了上一章建的 admin_restore。删完之后，可用 `SHOW DATABASES` 确认 `rookery` 是否真的已被删除。

接着，我们就可以开始恢复 `rookery` 了：

```
mysql --user=admin_restore --password < rookery.sql
```

以上是在命令行中用 `mysql` 执行 rookery.sql 里的语句。注意，小于号（`<`）使标准输入（`STDIN`）重定向到该文件，然后 `mysql` 便会抽取文件的内容并执行。最终，这条命令会创建 `rookery` 数据库及其所含的表，并往表中插入原本所有的数据。你可以在做完之后，登录 MySQL，切换到 `rookery`，用 `SHOW TABLES` 查看表的情况，并用 `SELECT` 看看数据是否都已恢复。这样练手是很必要的，而且也能让你对恢复整个数据库更有自信。

14.2.2 恢复表

用 dump 文件恢复整个数据库，会带来一个问题：可能会覆写了不相关的表。例如，假设

你不小心删掉了某个表，然后想恢复它。这时，该表所属数据库的其他表是正常的。如果最新的 dump 文件是在几个小时前生成的，但在这期间，其他表的数据有改动。如果用这个 dump 文件来恢复整个数据库，那么其他表的新增和修改的记录就会丢失。我想你应该不希望这种事发生吧。而如果每个表都分开备份，那恢复一个表就会很简单。不过，要维护一大堆 dump 文件又会很麻烦。好在我们还有一些方法，可以在使用整个数据库的 dump 文件时，限制只恢复单个表。下面就来看看这些方法。

1. 修改dump文件

我们在 14.1.2 节中已看到，数据库的 dump 文件是含有 SQL 语句的简单文本文件，这些语句可以用于创建数据库，然后分别恢复每个表及其数据。用数据库的 dump 文件恢复一个表的方法，就是修改 dump 文件。你可以将无关的语句去掉，只留下你要恢复的表的相关语句。

假设你有一个 dump 文件，里面只包含 `rookery` 数据库，而你因为不小心删掉或修改了部分数据，需要恢复 `conservation_status` 表。可以复制一份 rookery.sql，然后用普通的文本编辑器打开它，删掉创建其他表的部分，只保留临时变量的设置和恢复语句，以及 `conservation_status` 的部分。又或者，直接打开原本的 rookery.sql，只将需要的语句复制出来，粘到新的文本文件中。通过这些方法得到的新 dump 文件都应该是一样的，你可以用它来恢复表。

以下是经过截取的 dump 文件：

```
-- MySQL dump 10.14 Distrib 5.5.39-MariaDB, for Linux (i686)
--
-- Host: localhost   Database: rookery
-- ------------------------------------------------------
-- Server version      5.5.39-MariaDB

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,FOREIGN_KEY...=0*/;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Current Database: `rookery`
--

CREATE DATABASE /*!32312 IF NOT EXISTS*/ `rookery`
/*!40100 DEFAULT CHARACTER SET latin1 COLLATE latin1_bin */;

USE `rookery`;

-- [ snip ]
```

```
--
-- Table structure for table `conservation_status`
--

DROP TABLE IF EXISTS `conservation_status`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;

CREATE TABLE `conservation_status` (
  `conservation_status_id` int(11) NOT NULL AUTO_INCREMENT,
  `conservation_category` char(10) COLLATE latin1_bin DEFAULT NULL,
  `conservation_state` char(25) COLLATE latin1_bin DEFAULT NULL,
  PRIMARY KEY (`conservation_status_id`)
) ENGINE=MyISAM AUTO_INCREMENT=10
  DEFAULT CHARSET=latin1 COLLATE=latin1_bin;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `conservation_status`
--

LOCK TABLES `conservation_status` WRITE;
/*!40000 ALTER TABLE `conservation_status` DISABLE KEYS */;

INSERT INTO `conservation_status` VALUES
(1,'Extinct','Extinct'),
(2,'Extinct','Extinct in Wild'),
(3,'Threatened','Critically Endangered'),
(4,'Threatened','Endangered'),
(5,'Threatened','Vulnerable'),
(6,'Lower Risk','Conservation Dependent'),
(7,'Lower Risk','Near Threatened'),
(8,'Lower Risk','Least Concern'),
(9,NULL,'Unknown');
/*!40000 ALTER TABLE `conservation_status` ENABLE KEYS */;

UNLOCK TABLES;

-- [ snip ]

/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2014-09-15  6:48:27
```

这个 dump 文件会恢复 `conservation_status` 表。我用 `[ snip ]` 加了几个注释，说明截取了原 dump 文件的什么内容。另外，为了适应页面的宽度，我还加了一些回车。去掉这些

的话，该文件看起来应该与备份 conservation_status 所产生的 dump 文件一样。

这样做是可以的，只是比较无聊，而且你有可能截多或截少了某些语句。接下来我再介绍恢复单个表的其他方法。

2. 用临时数据库来做恢复

另一个恢复单个表的方法，就是修改 dump 文件中的数据库名。dump 文件一般包含一条 CREATE DATABASE 语句。如果把数据库名换成服务器上没有使用的名字，这样一旦执行该 dump 文件，就会创建一个新的数据库，而所有表和数据都会恢复到该数据库中。接下来，可以将需要的表从这个临时数据库复制到原数据库中。做完之后，可以删掉临时数据库。下面来看例子。

回到刚才的例子，假设你的 dump 文件中有整个 rookery 数据库，但你只想恢复其中的 conservation_status 表。如果你现在没有 rookery 的 dump 文件，那就用 mysqldump 创建一个，这样便可以跟着我一起做了。

首先，用 SHOW DATABASES 看看服务器上现存些什么数据库，以免临时数据库重名。接着，在文本编辑器中打开 dump 文件，找到开头的 CREATE DATABASE 语句附近的行。编辑这一部分，修改数据库名，如下所示：

```
--
...
-- Current Database: `rookery`
--

CREATE DATABASE /*!32312 IF NOT EXISTS*/ `rookery_backup`
/*!40100 DEFAULT CHARACTER SET latin1 COLLATE latin1_bin */;

USE `rookery_backup`;
...
```

可以看到，我把 CREATE DATABASE 和 USE 这两处的 rookery 改成了 rookery_backup。确实，要改的只有这两处。现在保存该 dump 文件并执行。使用一个具备 CREATE 权限的管理员账号，输入以下命令：

```
mysql --user=admin_restore --password < rookery.sql
```

执行过后，服务器上应该就多了 rookery_backup 数据库。现在，只要通过 mysql 客户端登录 MySQL，设 rookery_backup 为默认数据库，然后运行 SHOW TABLES 和几个 SELECT 语句，就会看到 rookery 的表和数据全都出现在这里了。现在就可以恢复需要的那个表了。

恢复表有两种方法。我们两种都试一下。首先，删掉 rookery 的 conservation_status 表。为此，你需要执行以下命令：

```
DROP TABLE rookery.conservation_status;
```

在 rookery 中新建一个 conservation_status 表。你可以在备份副本上做，这会用到 CREATE TABLE...LIKE（我们在 5.3.3 节中介绍过）。输入以下命令：

```
CREATE TABLE rookery.conservation_status
LIKE rookery_backup.conservation_status;
```

接着，从备份表中将数据复制到新建的表中：

```
INSERT INTO rookery.conservation_status
SELECT * FROM rookery_backup.conservation_status;
```

INSERT...SELECT 在 6.3.2 节中提到过。它会把 rookery_backup.conservation_status 里的所有行都复制到 rookery.conservation_status 里。执行过后，可以再用一个 SELECT 检查 rookery.conservation_status 是否已有数据。一切顺利的话，就可以删掉临时数据库了：

```
DROP DATABASE rookery_backup;
```

这种方法是可行的。不过，如果数据库很大，那么临时导入整个数据库就会很花时间。但话说回来，若真有这么大的数据库，那就应该分开备份表，这样才容易管理。采用这种做法，需要有 CREATE 和 DROP 权限来创建和删除数据库。

如果不想修改 dump 文件，可以试试下面介绍的方法。

3. 使用受限的用户账号

恢复单个表有一个很简单的方法，就是创建一个临时用户账号，使其只在要恢复的表上有操作权限。这样，当用这个账号来运行 dump 文件时，用于其他表的语句就会执行失败，而只有授权过的表才会被恢复。要建这样的账号，需要有 GRANT OPTION 权限（例如 root 就有）。下面继续用之前的例子（只恢复 conservation_status 表），来看看这种方法的具体步骤。



其实这种方法有点危险。如果不清楚给这个账号哪些权限，或者用 root 而不是受限的用户账号不小心从 dump 文件中恢复数据，那么就有可能会覆盖所有备份到 dump 文件中的数据库。所以，要很小心。

为了更好地看出此法的妙效，我们在恢复数据之前除了要删掉 conservation_status 表，还要改动一下其他表的数据，以验证恢复后这些改动是否还存在。我们用第 13 章创建的 admin_boss 来做：

```
mysql --user=admin_boss --password \
      --execute "DROP TABLE rookery.conservation_status;
                 INSERT INTO rookery.birds (common_name,description)
                 VALUES('Big Bird','Large yellow bird found in New York');

                 SELECT LAST_INSERT_ID();"
```

以上语句应该能删掉 conservation_status。而为了验证恢复会不会影响到 conservation_status 以外的其他表，我们在 birds 表中新增了一行，并用最后一句 SELECT LAST_INSERT_ID() 查回该行的 ID。你甚至可以登录 MySQL，检查是否真的删掉了 conservation_status，并用刚才的 ID 查查 birds 是否真的新增了一行。如果没问题的话，就可以继续了。

现在，创建受限的管理员账号 admin_restore_temp。输入 GRANT 语句：

```
GRANT SELECT
ON rookery.* TO 'admin_restore_temp'@'localhost'
```

```
IDENTIFIED BY 'its_pwd';

GRANT ALL ON rookery.conservation_status
TO 'admin_restore_temp'@'localhost';
```

这先使 admin_restore_temp 在 `rookery` 所有表上有 `SELECT` 权限，而后再让它单独在 `conservation_status` 表上有 `ALL` 权限。于是，当你恢复包含 `rookery` 所有表的 dump 文件时（使用 admin_restore_temp），便只有 `conservation_status` 会被替换。

当用这个用户账号执行 dump 文件时，因为无权操作其他表，所以如果它想替换其他表，MySQL 就会报错。正常来说，dump 文件的运行会终止。为了能顺利进行，可以用 `--force` 选项来忽略报错信息。

现在，使用以下命令恢复表：

```
mysql --user admin_restore_temp --password --force < rookery.sql
```

这应该没有问题。请自行登录 MySQL，并检查 `conservation_status` 是否已恢复。然后，用 `SELECT` 检查 `birds` 表新增的 Big Bird 是否还在。如果在，那表示在恢复 dump 文件时没有覆盖 `birds` 表，而其他表也应该没受影响。

14.2.3　只恢复某些行或列

一般来说，恢复整个数据库或整个表是很少见的。我们很少会误删整个数据库或整个表，或一下子改错了表中的所有行。更多的情况是，有人误删了表中的某行或误改了某列，却无法撤销操作。如果其他行的改动无误，而你只有这些改动前的备份，那么用这个备份来恢复整个表就因小失大了。这时候，应该只恢复某一行或某一列。

这可以通过上一节介绍的临时数据库来做恢复。上一节介绍了如何修改 `rookery` 数据库的 dump 文件，这样 MySQL 就能把这个数据库导入新的临时数据库（`rookery_backup`）。如果用这种方法，就能用带 `WHERE` 的 `INSERT...SELECT`，只选取要恢复的行。下面我们就来试试。

假设有人不小心删掉了 `birdwatchers` 数据库的 `humans` 表的一条会员记录（Lexi Hollar），并清空了另一个会员（Nina Smirnova）的邮件地址。为了模拟这个情境，我们先做一个 `birdwatchers` 的备份，然后再删掉上述两条记录：

```
mysqldump --user=admin_backup --password --lock-tables \
          --databases birdwatchers > birdwatchers.sql

mysql --user=admin_restore --password \
      --execute "DELETE FROM birdwatchers.humans
                 WHERE name_first = 'Lexi'
                 AND name_last = 'Hollar';

                 UPDATE birdwatchers.humans
                 SET email_address=''
                 WHERE name_first = 'Nina'
                 AND name_last = 'Smirnova'"
```

执行以后，登录 MySQL，确认是否在 humans 表中删掉了 Lexi Hollar 这个名字和 Nina Smirnova 的邮件地址。尽管你对这些改动很确信，还是应该确认一下。完成这些过程对你是有益的，这会让你在后面的恢复工作中更有信心。

接着，把 birdwatchers 数据库导入临时数据库。编辑刚创建的 birdwatchers.sql，并查找与该数据库相关的 SQL 语句，应该只有 CREATE DATABASE 和 USE。把数据库名改成 birdwatchers_backup（假设服务器上没有这样的数据库名）。保存 dump 文件并退出。用以下命令导入这个 dump 文件：

```
mysql --user=admin_restore --password < birdwatchers.sql
```

执行以后，登录 MySQL，用 SHOW DATABASES 看看 birdwatchers_backup 是否已创建。如果建好了，就用以下命令来恢复 humans 表中的数据：

```
REPLACE INTO birdwatchers.humans
SELECT * FROM birdwatchers_backup.humans
WHERE name_first = 'Lexi' AND name_last = 'Hollar';

UPDATE birdwatchers.humans
SET email_address = 'bella.nina@mail.ru'
WHERE name_first = 'Nina' AND name_last = 'Smirnova';
```

这就恢复了 Lexi Hollar 的整条记录，以及 Nina Smirnova 的邮件地址，而且对数据库中的其他行或表毫发无伤。注意，这里用了 REPLACE，而不是 INSERT。如果 MySQL 找到了和 WHERE 子句匹配的行，并且其 human_id 又相同，那备份表里匹配的行就会替换掉该行；否则，就会插入备份新的一行。无论怎么做，恢复后的 human_id 都是原本的 human_id。这意味着参考那一行的其他表都会有正确的 human_id。顺便说一下，给 mysqldump 加上 --replace 选项，就可以使生成的 dump 文件不用 INSERT，而用 REPLACE。

恢复完后，用 DROP DATABASES 删除 birdwatchers_backup 数据库。

这个方法很适用于恢复某些行或列，特别是在你只想修复某个人的误操作，而又不想影响到其他人的时候。它十分简单，通常不会花费太多时间。你可以在 WHERE 中加上各种各样的条件，以限定恢复的范围。如果你想成为优秀的数据库管理员，那就要好好掌握这一技能。当你可以毫不费劲地恢复数据库时，用户会非常崇拜你。

14.2.4　用二进制日志来做恢复

前几节介绍的恢复方法大多是常用方法。有时你需要更精确的方法，如前面讲到的恢复单行和单列。当你恢复指定的行时，就会用到这个方法，而丢失的数据保存在 dump 文件中。然而，假设你要恢复的是备份后才发生的数据改动，可能很多人都会觉得做不到。其实，只要你理解 MySQL 的二进制日志，并足够谨慎便可办到。可以用二进制日志来将数据恢复到指定的时间点（这些数据是在最近一次创建 dump 文件时产生的）。这种做法被称为时间点恢复。

时间点恢复的前提条件是，你开启了二进制日志。如果等到出问题时才开启，那将无济于事。想知道二进制日志是否开启，可以用以下命令检查：

```
SHOW BINARY LOGS;

ERROR 1381 (HY000): You are not using binary logging
```

如果出现以上报错信息，就需要去开启二进制日志。

> 开启二进制日志可能会有安全隐患。所有执行过的、修改数据的 SQL 语句都会被记录在二进制日志中，其中可能包括一些敏感数据（如信用卡号码）和各种密码。因此，需要确保不是人人都能访问日志文件及其所在目录，并且日志当中不要记录含有用户账号密码的 `mysql` 数据库的相关操作。如要排除日志中的某些数据库，可以用 `--binlog-ignore-db` 选项指定。

开启二进制日志的方法如下：修改 MySQL 配置文件（具体看你的是什么系统，可能是 my.cnf 或 my.ini），在 `[mysqld]` 那一部分中，加入以下语句：

```
log-bin
binlog-ignore-db=mysql
```

注意第一行是不带等号或值的。第二行指示 MySQL 忽略 `mysql` 数据库的改动。在配置文件中加完这些内容之后，记得重启 MySQL，以使配置生效。之后，再次登录 MySQL，检查二进制日志是否已开启。这次，我们用 `SHOW MASTER STATUS`：

```
SHOW MASTER STATUS;

+--------------------------+----------+--------------+------------------+
| File                     | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+--------------------------+----------+--------------+------------------+
| mysqlresources-bin.000001 |      245 |              | mysql            |
+--------------------------+----------+--------------+------------------+
```

它显示了当前在用的日志文件的名字，以及被忽略的数据库。

既然 MySQL 正在记录二进制日志中的所有 SQL 语句，那就可以做时间点恢复了。为了体验这个过程，我们需要登录 MySQL，然后往表中插入大量数据。为了简单起见，我们直接使用 MySQL 资源站（http://mysqlresources.com/files）的 birds-simple.sql 和 birds-simple-transactions.sql。前者会在 `rookery` 数据库中增加含有数据的 `birds_simple` 表。后者会先往该表插入一些行，然后通过一条 SQL 语句修改某些行，以模拟误改数据，最后再插入几行。而我们要做的练习是：仅恢复误改前和误改后所产生的行。如果你要跟着我做，请先下载那两份 dump 文件，然后到文件所在目录执行以下命令：

```
mysql --user=admin_restore --password --database=rookery < birds-simple.sql

mysql --user=root --password --silent \
      --execute="SELECT COUNT(*) AS '' FROM rookery.birds_simple;"
```

你可能会注意到，我在第一条命令中加了 `--database` 选项，把它设置为 `rookery`。当我生成 dump 文件时，只 dump 了 `birds_simple` 表。结果，这个 dump 文件中没有 `USE`，而且表名前也不带数据库名。所以，需要加上 `--database=rookery`，使 MySQL 在 `rookery` 数据库中执行 dump 文件中的所有语句。而如果一切顺利，第二条命令应该会告诉你 `birds_`

simple 表现在有 28 892 行。

接着，用 birds-simple-transactions.sql 来破坏该表中的数据（增加和删除许多行）：

```
mysql --user=admin_restore --password \
      --database=rookery < birds-simple-transactions.sql

mysql --user=root --password --silent \
      --execute="SELECT COUNT(*) AS '' FROM rookery.birds_simple;"
```

birds-simple-transactions.sql 包含了带 WHERE 的一些 DELETE 语句，用来删除许多行。它还有一些 INSERT 语句，用来新增一些行。最后结果应该会比刚才少了 296 行。

接下来，我们准备逐步进行时间点恢复。为了把所有数据恢复到指定的时间点，我们先以最近一次备份为起点，即 birds-simple.sql：

```
mysql --user=admin_restore --password \
      --database=rookery < birds-simple.sql
```

这就使 birds_simple 表恢复到生成 birds-simple.sql 的那个时间点。你可以登录 MySQL，计算 birds_simple 表的行数。现在应该是 28 892 行了。

下一步是获取此状态后被执行过的 SQL 语句。如果数据库很繁忙，一直不停地在执行 SQL 语句，那要找出其中某一条，可能会稍微麻烦。因此，如果想结合使用 mysqldump 和 mysqlbinlog，就应该在做备份时用 mysqldump 刷新日志。我在生成 birds-simple.sql 时就加了 --flush-logs 选项。现在我们需要从当前日志文件的开头，将数据恢复到执行 DELETE 语句的那个时间点。可以从二进制日志中确定那个时间点。

接着，只需用 mysqlbinlog 抽取当前二进制日志的内容并将其保存到文本文件中，就能方便地定位哪些语句出问题了。

1. 翻查二进制日志

翻查之前，你得先知道包含 SQL 语句的二进制日志的名字和位置。可以用 SHOW MASTER STATUS 来查名字，而位置通常是在 data 目录中，具体可以用 SHOW VARIABLES 来查：

```
SHOW MASTER STATUS;

+---------------------------+----------+--------------+------------------+
| File                      | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+---------------------------+----------+--------------+------------------+
| mysqlresources-bin.000002 |  7388360 |              | mysql            |
+---------------------------+----------+--------------+------------------+

SHOW VARIABLES WHERE Variable_Name LIKE 'datadir';

+---------------+--------------+
| Variable_name | Value        |
+---------------+--------------+
| datadir       | /data/mysql/ |
+---------------+--------------+
```

第一条语句显示当前二进制日志文件的名字是 mysqlresources-bin.000002。之所以名字中改

成了 2，是因为我做 birds-simple.sql 时，加上了 `--flush-logs`。至于第二条语句，则显示日志文件放在 /data/mysql/ 目录中（当然你也可以去检查一下它是否真的在那里）。知道位置和名字后，就可以用以下命令来抽取我们需要的内容了：

```
mysqlbinlog --database=rookery \
          /data/mysql/mysqlresources-bin.000002 > recovery-research.txt
```

如你所见，我用了 `--database` 选项，限定 `mysqlbinlog` 只抽取与 `rookery` 有关的语句。否则，抽取的文件就会包含其他数据库的活动。另外，因为在我的服务器上，数据库多达二十几个，其中某些还被频繁地存取，所以，为了使恢复过程更简单，以及避免不小心重写了其他数据库，我就作了这样的限定。

接着，只需指定二进制日志文件的路径和名字，以及重定向的位置（`> recovery-research.txt`），即可将 `mysqlbinlog` 的结果导入该文件。

2. 从二进制日志中抽取语句并执行

导出完成以后，用文本编辑器打开该文件，查找 `DELETE` 语句。因为我们只做过两次 `DELETE` 操作（我已截取如下），所以恢复起来应该很简单：

```
# at 1258707
#140916 13:10:24 server id 1 end_log_pos 1258778
Query thread_id=382 exec_time=0 error_code=0
SET TIMESTAMP=1410887424/*!*/;
SET @@session.sql_mode=0/*!*/;

BEGIN
/*!*/;

# at 1258778
#140916 13:10:24 server id 1 end_log_pos 1258900
Query thread_id=382 exec_time=0 error_code=0
use `rookery`/*!*/;
SET TIMESTAMP=1410887424/*!*/;

DELETE FROM birds_simple WHERE common_name LIKE '%Blue%'
/*!*/;

# at 1258900
#140916 13:10:24 server id 1 end_log_pos 1258927 Xid = 45248

COMMIT/*!*/;

...

# at 1284668
#140916 13:10:28 server id 1 end_log_pos 1284739
Query thread_id=382 exec_time=0 error_code=0
SET TIMESTAMP=1410887428/*!*/;
SET @@session.sql_mode=0/*!*/;
BEGIN
/*!*/;
```

```
# at 1284739
#140916 13:10:28 server id 1 end_log_pos 1284862
Query thread_id=382 exec_time=0 error_code=0
SET TIMESTAMP=1410887428/*!*/;
DELETE FROM birds_simple WHERE common_name LIKE '%Green%'
/*!*/;

# at 1284862
#140916 13:10:28 server id 1 end_log_pos 1284889 Xid = 45553
COMMIT/*!*/;
```

有人可能会觉得日志很混乱，但其实只要你理解二进制日志的结构以及事务这些概念，就不会这样认为。

二进制日志的每个条目都以两行注释（以 # 开头的行）开始。第一行注释的号码是位置号（在 at 后面），可用来指定恢复到哪一点。而第二行则记录了语句执行的时间及其他信息。最后，条目以 /*!*/; 作为结尾。

而所谓事务，通常就是指一组同时执行且彼此相关的 SQL 语句。它只在事务性表（比如 InnoDB）上可用，而在非事务性表（比如 MyISAM）上不可用。同一事务中的语句，只要还未提交，就可以全部撤销或回滚。而为了能如实恢复数据，二进制日志会保留事务语句。也因为这一特性，我们确实需要看看日志中事务的组件。

BEGIN 和 COMMIT 是事务开始和结束的标志。其中，COMMIT 会把该范围中的语句提交，而一旦提交，就不能回滚或撤销了。从二进制日志开头往下看，你会发现，BEGIN 后面紧跟着第一个 DELETE。所以，DELETE 是被夹在 BEGIN 和 COMMIT 之间的。

第一个 DELETE 的位置号是 1258778。但是，这已经进入事务之中了。要想查出事务的范围，还是要往上看：

```
# at 1258707
#140916 13:10:24 server id 1 end_log_pos 1258778 Query thread_id=382
```

到达 BEGIN 条目了，它的位置号是 1258707，日期和时间是 140916 13:10:24（即 2014 年 9 月 16 日下午 1 点 10 分 24 秒）。因此，我们已掌握了第一个 DELETE 所在事务的位置号和开始时间。细看一下，你还会发现 end_log_pos 后面还有一个数字，它指的是下一条目的位置号（1258778）。不要被它迷惑。其实，位置号不是简单地递增的，它记录的是条目在日志中的位置。

因为我们要恢复第一个 DELETE 所在事务之前的语句（也就是说直至 1258707），所以，可以修改这个导出的日志文本文件，去掉不需要的事务语句，然后用 mysql 客户端读取并运行那些剩下的语句。但其实，还有更简单的方法：用 mysqlbinlog 再一次导出事务，但只导到 1258707 为止。做法如下：

```
mysqlbinlog --database=rookery --stop-position="1258707" \
             /data/mysql/mysqlresources-bin.000002 |
             mysql --user=admin_restore --password
```

这样，导出的文件就只包含 1258707 之前的语句了。

用这份文件，我们就恢复好 DELETE 之前的操作了。而接着，我们还需要恢复第二个 DELETE

之后的那些事务。

在第二个 DELETE 之后的 COMMIT 处，可以看到它的 end_log_pos 是 1284889。这是第二次删除数据之后的第一个事务的起点，而我们就是要恢复它之后的动作。我们无需指定终点的位置号，只要用 --to-last-log 选项，即可一直导出，直到日志的结尾。顺便说一下，如果之后又刷新或增加过二进制日志，那么后面文件的内容也会被输出。总之，使用如下命令：

```
mysqlbinlog --database=rookery  --start-position="1284889" --to-last-log \
            /data/mysql/mysqlresources-bin.000002 |
            mysql --user=admin_restore --password
```

这就能恢复 DELETE 之后的所有操作了。因为通过日志文件的位置号来定位到指定的事务，所以此法能非常精准地控制恢复范围。而除了位置号，还可以用开始时间和停止时间来进行时间点恢复，用到的选项是 --start-datetime 和 --stop-datetime。而如果想做出上例的效果，可以这样：

```
mysqlbinlog --database=rookery --stop-datetime="140916 13:10:24" \
             /data/mysql/mysqlresources-bin.000002 |
             mysql --user=admin_restore --password

mysqlbinlog --database=rookery --start-datetime="140916 13:10:29" --to-last-log \
            /data/mysql/mysqlresources-bin.000002 |
            mysql --user=admin_restore --password
```

很好理解，第一条语句要使用的停止时间是第一个 DELETE 事务的开始时间，而第二条语句的开始时间则是第二个 DELETE 事务结束的后一秒。这应该是没什么问题的，不过考虑到一秒钟也可以执行很多动作，所以，想更精确的话，还得用位置号。

二进制日志也可用于做备份，即 MySQL 复制。所谓复制，就是让另一个服务器（被称为从服务器）持续读取主服务器的二进制日志。由此，从服务器便成为了主服务器的复制体。当需要另外再做备份时，可以直接用从服务器来做。只需暂停其对主服务器日志的读取，然后备份从服务器上的数据库。做完后，重启读取即可。因为读取的是二进制日志，所以从服务器很快就可以又与主服务器同步。不过，此话题已超出本书范围。如果想了解或学习如何解决复制问题，可以参考我的另一本书 *MySQL Replication: An Administrator's Guide to Replication in MySQL*（A Silent Killdeer Publishing, 2010）。

14.3 制定备份策略

学会备份和恢复数据库没错，但这些技能也只在你需要定期、有效地做备份时才有意义。若是你的恢复会损坏数据库，或导致更多的数据损失，或其过程太过缓慢，那备份的价值就大打折扣了。要想成为优秀的数据库管理员，你还需要学会制定备份策略，并将其落实执行。

备份策略需要留底，即使仅有你一个人看。而且，它应该涵盖备份和恢复的各个方面。制定备份策略时，需要考虑实际情况，根据各数据库的价值、数据的敏感程度，以及其他因素而定。举个例子，假如你的数据库仅用作个人网站的数据保存，不用来赚钱，也与其他人没有利害关系，并且其内容不常更新，那么，你的策略可以是一周做一次完整的备份，并让每个备份最少保留一个月。而如果你是一个大型网站的数据库管理员，该网站有数百万行数据置于多个表中，每日有数千人访问，并且存有价值巨大的一些信用卡号码，那么你的备份策略要做得更加精细才行。在这种情况下，需要考虑到数据安全、访问阻塞和恢复速度的问题。而本节接下来介绍的例子，则处于这两种极端之间。通过例子，你应该会明白制定策略时要思考些什么。

要制定策略，首先就是搞清楚自己要负责哪些数据库和表。本例将用到贯穿全书的 `rookery` 和 `birdwatchers` 这两个数据库。现假设我们的观鸟网站已运行了多年，吸引了众多会员。为模拟这个情况，我特意增加了大多数表的行数，并删除了一些临时用的表。而这大量的数据也使制定策略显得十分必要。最后，为明确需要备份哪些表，我将它们按数据库分组，并以字母序列出，如表 14-1 所示。

表14-1：评估用于备份策略的数据库

表	行　数	更　新	繁　忙	敏　感
rookery				
bird_families	229		√	
bird_images	8			
bird_orders	32		√	
birds	28 892		√	
birds_bill_shapes	9			
birds_body_shapes	14			
birds_details	0			
birds_habitats	12			
birds_wing_shapes	6			
habitat_codes	9			
birdwatchers				
bird_identification_tests	3201	√	√	
bird_sightings	12 435	√	√	
birder_families	96			√
birding_events	42	√		
birding_events_children	34			√
humans	1822	√	√	√
prize_winners	42		√	
survey_answers	736			
survey_questions	28			
surveys	16			

该表所列的关于两个数据库的参数，对如何制定策略至关重要。这些参数是：每个表的行数、更新情况（数据和结构是否经常改变）、访问热度和敏感程度。当然，你自己制定策略时，可以考虑其他因素，但此例暂时只考虑这几个方面。

一般来说，对于那些不常更新的表，是否每日备份无关紧要。而那些经常更新的表则要每日备份，并选择在不太繁忙的时间进行。如果某个表含有敏感数据（比如会员的个人或家庭信息），则要用特殊的用户账号来执行备份，并将备份文件放到更安全的目录下。同时，每周做一次完整的备份，因为 dump 文件也会包含敏感数据，所以也要将其放在刚才提到的安全目录下。

记住这些之后，就可以开始计划各数据库和表的备份时间和存放目录了。如表 14-2 所示，因为我先将备份按数据库分组，然后再按保密程度和用途来分，所以，有些备份文件会包含整个数据库的所有表，而有些则只包含部分。此外，我还在右侧列出每个备份的产生周期、具体时间、是否需要保密，以及是否离线存储。

表14-2：备份计划

备　份	频率	日期	时间	是否保密	是否离线存储
rookery 整个数据库 所有表 (rookery-yyyy-mmm-dd.sql)	每周	周一	8:00	否	是
rookery 鸟类信息 birds、bird_families、bird_orders (rookery-class-yyyy-mmm-dd.sql)	每日	每日	9:00	否	否
birdwatchers 整个数据库 所有表 (birdwatchers-yyyy-mmm-dd.sql)	每周	周一	8:30	是	是
birdwatchers 观鸟者相关信息 humans、birder_families、birding_events_children (birdwatchers-people-yyyy-mmm-dd.sql)	每日	每日	9:30	是	否
birdwatchers 活动信息 bird_sightings、birding_events、bird_identification_tests、 prize_winners、surveys、survey_answers、survey_questions (birdwatchers-activities-yyyy-mmm-dd.sql)	每日	每日	10:00	否	否

注：时间按 G.M.T.。含敏感信息的备份需要用特殊的管理员账号来操作，并存放在安全的目录下。而且，有些需要离线存储。

注意，上表计划每周做一次全数据库备份，每个数据库单独有一个 dump 文件。你可能想将它们合在一起，但我认为分开会更方便我们之后恢复某一个数据库。

而除了每周一次的全数据库备份，我们还规定了让某些经常更新的表每天备份，不管是内容还是结构都要备份。虽然有些表不常更新，本不需要频繁备份，但因为它们也不是很大，所以每天备份一次也没什么问题。有些人可能觉得，每天都备份全数据库是最简单的。但如果数据库非常庞大，或者你担心安全和性能问题，那这就不是最佳选择了。所以，为了让你懂得从各个角度来考虑问题，这里的例子针对不同的表有不同的方案。

在我们这个虚构的观鸟网站中，有很多来自欧洲和美国的会员。因为观鸟对大多数人来说只是一种爱好，所以人们都是在晚上才访问我们的网站。因此，我们把时间定在 G.M.T. 的早上。等到伦敦时间早上 8 点，就启动第一个备份命令，而旧金山此时正是午夜。换句话说，我们是在美国的深夜时分开始做备份，因为这应该是数据库流量最少的时段。

我们将所有的备份保留在本机和两个单独的服务器上。通过内网，使用 `cron` 把 dump 文件自动复制到第二个服务器上。此外，考虑到火灾或其他灾难可能毁坏同一栋楼的服务器，所以，我们还要把每周的全数据库备份复制到 DropBox 或 Google Drive 之类的云服务器上。

除了备份计划，还需要备份验证计划（见表 14-3），以确保备份得以正确执行。该计划包括：查看备份文件是否生成，并测试生成的文件能否用来做恢复。这个计划还能锻炼我们的恢复技能。其实我已说过多次，一旦发生需要恢复数据这种紧急情况，你就得马上投入，做该做的事。到发生紧急情况时才学习是不太可能的，所以，平时就要进行操练。

表14-3：备份验证计划

备　份	验　证	库恢复测试	表恢复测试	行恢复测试	保留期限
`rookery` 整个数据库	每周	每月	无	半月	两月
`rookery` 鸟类信息	每周	无	半月	半月	一月
`birdwatchers` 整个数据库	每周	每月	无	半月	两月
`birdwatchers` 观鸟者相关信息	每周	无	半月	半月	一月
`birdwatchers` 活动信息	每周	无	半月	半月	一月

注：定期检查备份文件。此外，出于测试和练习的需要，还会在测试环境中，定期进行数据库、表、行的恢复。

现在，仔细看看该计划。我们的检查将每周进行一次：检查该周要做的 dump 文件到底有没有生成，以及文件中到底有没有我们需要的表。具体来说，就是检查文件是否存在，大小如何。另外，可以用文本编辑器打开它们，看看内容对不对。也可以用 `grep` 来抽取含 `CREATE TABLE` 的行，以得到表名。例如，用以下命令就可以截取出 rookery.sql 所含的表：

```
grep 'CREATE TABLE' rookery.sql | grep -oP '(?<=CREATE\ TABLE\ \`).*(?=\`)'

bird_families
bird_images
bird_orders
birdlife_list
birds
birds_bill_shapes
birds_body_shapes
birds_details
birds_habitats
birds_wing_shapes
conservation_status
habitat_codes
```

表 14-3 接下来的三列则与测试和恢复数据有关。全数据库恢复是每月测试一次。你可以在测试环境中进行，恢复过后，在正式环境和测试环境都执行查询，对比看看结果有没有出入（应该略有不同）。

其他 dump 文件是基于表的。这些表要么是经常更新的，要么对于我们的观鸟网站来说十分重要。所以基于表的恢复测试，我们半月做一次。此外，鉴于行恢复是比较常见的需求，对于所有备份，我们都进行半月一次的行恢复。知道怎样从所有 dump 文件中恢复指定的数据，是非常重要的。只要多加练习，在需要恢复丢失的少量数据时，就不用太担心。

最后一列是关于备份文件保留时长的。目前的计划是全数据库备份最多保留两个月，表备份则保留一个月。当然，这个随个人需要而定。有人甚至会把每个月的 dump 文件刻到光盘上，以便保存好几年。

我们的备份策略大致就如表 14-2 和表 14-3 所示。其中，表 14-2 规定的是备份什么、何时备份，以及备份到哪里。而表 14-3 规定的则是检查备份是否成功、恢复的周期，以及备份的保存期限。除了这些以外，影响备份策略的因素还有很多，你可以自己加到表中。但无论怎样，本节的思路都值得参考。

14.4　小结

现在，你应该很清楚备份的重要性了。做好备份，便能摆脱数据丢失的危机。而熟练掌握各种恢复技巧，则能让你更灵活地解决问题。此外，为了让你对备份和恢复的学习及锻炼不至于白费，应该制定合理的备份策略，并严格遵守之。

如本章开头所述，备份工具不止一种，而且方法也是多样的（甚至可以用 MySQL 复制来做）。其中 `mysqldump` 是最简单的，在有些情况下，它还是最佳选择。如果你是数据库管理员，就应该好好掌握这一工具，并学会如何用 dump 文件做恢复。最后，为了练手，请完成下一节的习题。

14.5　习题

以下题目可使你更加熟悉如何用 `mysqldump` 生成备份，以及用生成的备份文件进行恢复。因此，你应尽力完成它们。其中某些题是有难度的，一下子没做出来的话，也请多尝试。

(1) 为了让接下来的练习不出什么问题，先用 `mysqldump` 做两个备份。一个是备份所有数据库，另一个是备份 `rookery` 和 `birdwatchers`（到同一个 dump 文件中）。但注意，接下来的练习不要使用这两个备份文件。保留它们，万一出错了，你需要用它们做恢复。

(2) 参考表 14-2。它包含一列需要定期创建的备份，有两个全数据库备份，以及三个基于表的备份。用 `mysqldump` 创建这五个备份，要求生成的 dump 文件按表中的命名规范来取名字。

(3) 重做第 2 题，但用五个 shell 脚本来分别产生这五个备份。备份文件的名字要符合命名规范，并根据当前日期来生成。关于如何写脚本，可以参考 14.1.4 节中的例子。直接在这个脚本上修改，或自己重写。当然也可以使用 shell 以外的编程语言。

写完后，执行它们，看看能不能生成那五个备份，以及名字都对不对。如果环境允许，就将发起这些脚本的命令加到 crontab 中（或其他调度工具中），让它们定期自动运行（最好能尽快运行一次）。然后看它们能否按时备份。试完之后，便可从 crontab 中清除。

(4) 修改上一题的脚本，使它们能按表 14-3 所定的保留期限来清理旧备份。你可以这样制造测试数据：把生成的备份文件复制出几份，并修改日期后缀，使一些文件“过期”，一些“未过期”。

然后，再执行脚本，看看有没有删除过期备份（可能需要试几次）。

(5) 登录 MySQL，用 DROP TABLE 删除 birds_bill_shapes 和 birds_body_shapes。

接着，用第 2 题所生成的 rookery 全数据库备份，来恢复这两个表。操作完后，登录 MySQL 并检查是否恢复成功，数据是否存在。

(6) 登录 MySQL，将 birds 表中含 Parrot 字眼的 common_name 全都更新成 NULL。应该有 185 行。

把 rookery.sql 复制一份，命名为 rookery_temp.sql。然后将其中的数据库名字改为 rookery_temp。若忘记了怎么改，可参考 14.2.3 节。

然后，用 rookery_temp.sql 创建 rookery_temp 数据库，并用 UPDATE，依据 rookery_temp.birds，恢复 rookery.birds 中俗名含有 Parrot 的行。

(7) 开启二进制日志这一功能（开启方法请参考 14.2.4 节。开启后，记得重启 MySQL）。然后用 mysqldump，带上 --flush-logs 选项，只备份 rookery 的 birds 表。

完成上述要求后，登录 MySQL，用 DELETE 删除 birds 表中俗名带 Gray 的行，再插入几行（可自己编一些俗名，并让其他栏位留空）。

然后，用刚才备份的 dump 文件来恢复 birds 表。再用 14.2.4 节所提到的时间点恢复方法，配合 mysqlbinlog，恢复（当前）二进制日志中，删除 Gray 鸟的 DELETE 语句前的所有事务。（需要找出该 DELETE 在日志中的位置号。）

接着，用这个位置号恢复该 DELETE 后，直到二进制日志结尾的所有事务。

都恢复完后，登录 MySQL，检查数据是否恢复。完成所有习题后，如果不再需要记录事务，记得停用二进制日志。

第 15 章

批量导入数据

本章将介绍各种批量导入数据的方法。有了这些方法，当你想用 MySQL 或 MariaDB 来替换使用另一种数据库系统（或另一种存储格式）的数据库时，或者当你想从某种非数据库软件将数据提取到 MySQL 或 MariaDB 中时，就无需手动录入数据，而直接选择批量导入。

但这里还有一些要求。如果数据来源于另一种应用程序，需要确保从它导出的数据符合 MySQL 的读取习惯。例如，将每项数据用特定的字符分隔好，并放在文本文件中，这样是可以接受的。无论数据量有多大，只要整理好它的格式，并保存在文本文件中，你就可以用 `LOAD DATA INFILE` 语句来导入数据。

这不是难事，但第一次导入大量数据的过程，可能会让人感到紧张，甚至望而却步。这会是将数据迁移到 MySQL 或 MariaDB 的一个障碍。要想做好一次完美的导入，确实有很多细节需要注意，特别是当你想更进一步把这个过程自动化时，要考虑的就更多了。另外，如果你是要把数据导入托管站，那还要对限制条件有些心理准备。而这些问题都会在本章讨论。

15.1　准备导入

要想把数据导入 MySQL 或 MariaDB，需要将数据整理成一种它们读得懂的格式。事实上，只要是把数据保存在文本文件中，并以某种形式分隔每个数据项，就已经符合它们的要求了。如果你的数据不是这样，那最简单的方法就是把它导回其来源软件，然后再导出成内容带有分隔符号的文本文件。一般的软件应该都有这样的功能。而它们导出的文件通常以逗号来分隔每个域，并通过换行来分隔每条记录。有些软件允许自定义分隔符。可以选择用竖线来分隔每个域（因为竖线不太会出现在实际数据中），然后通过换行来分隔每条记录。

现在我们先获取一个与 rookery 数据库有关的大数据文件。我们知道，康奈尔大学在鸟类学研究方面非常有名，该校甚至还通过自己的出版社发行过相关的书籍。而其中一本就是由 James F. Clements 编著的《世界鸟类名录》（*The Clements Checklist of Birds of the World*）。我们可以从其网站（http://www.birds.cornell.edu/clementschecklist/）获取此书的鸟种列表。它是一个逗号分隔值（Comma-Separated Values，CSV）文件，每年八月都会更新一次，并免费提供给人们或组织使用，以促进鸟类研究和鉴赏事业的发展。

假设我们想拿这份最新的鸟种表与我们的 birds 表对比，看有没有什么新的鸟种。这听起来好像很麻烦，但其实并不难。为此，我们需要从康奈尔大学的网站或 MySQL 资源站（http://mysqlresources.com/files）下载该表。在下文的示例中，我用到的是 Clements-Checklist-6.9-final.csv。

> 如果通过 FTP 把数据文件上传到服务器，记得要用 ASCII 模式来传，不要用二进制模式。如果文件本身就包含二进制字符或二进制换行符，也一样不能被 MySQL 读取。

下载好康奈尔大学的数据文件之后，用文本编辑器打开它，看看里面的内容。你需要关心的是，它的每条记录、每一个域以什么符号来做分隔。以下是我下载的文件的部分内容：

```
sort 6.9,Clements 6.9 change,2014 Text for website,
Category,Scientific name,English name,Range,
Order,Family,Extinct,Extinction Year,sort 6.8,sort 6.7,page 6.0,,,,,

...

4073,new species,"Walters (1991) and Cibois et al. (2012) proposed
recognition of Prosobonia ellisi Sharpe 1906, with English name
Moorea Sandpiper and range ""extinct;
formerly Moorea (Society Islands)"".",
species,Prosobonia ellisi,Moorea Sandpiper,extinct;
formerly Moorea (Society Islands),
Charadriiformes,Scolopacidae (Sandpipers and Allies)
  ,1,xxxx,,,addition (2014),,,,,

...

6707,new species,"Robb et al. (2013) describe a new species of owl, Omani Owl
(Strix omanensis), from the Arabian Peninsula, with range
""central Al Hajar mountains, northern Oman"".
Position Omani Owl immediately following Hume's Owl (Strix butleri).",
species, inStrix omanensis,Omani Owl,"central Al Hajar mountains, northern Oman",
Strigiformes,Strigidae (Owls),,,,,addition (2014),,,,,

...
```

该 CSV 文件大概有 32 000 行，而我只截出某些我感兴趣的行。并且，因为每条记录都很长，所以我只好手动换行，使它符合页宽。

第一行是每个域的名字。你或许没能一下子看懂它们，因为它们有些用于与早期发布的列表进行对照。比如，第一个域，sort 6.9，其实是指每一行的标识号。而后面的 sort 6.8

和 sort 6.7 就是该记录在旧表中的标识号。虽然此表还有很多域，但对于本章要演示的例子来说，我们只需关注 Clements 6.9 change、Scientific name、English name、Order 和 Family 这几个域。

Clements 6.9 change 用于描述该记录与上一份表相比有什么变化。而现在，我们想看的是 new species。

我所截出的两条记录都需要导入。其中 4073 那条记录就是一个新鸟种，叫 Prosobonia ellisi 或 Moorea Sandpiper。但很可惜的是，这种鸟已经灭绝了。尽管有些鸟种已经灭绝，但鸟类学家还是会把它们都记录下来。同样，为了遵守这一规则，我们也要把它加到 birds 表中（虽然不会再有人能观察到这种鸟）。而 6707 是另一个新鸟种，叫 Strix omanensis 或 Omani Owl，来自阿拉伯半岛。还好，它仍存活着。

在导入之前，先把这个 CSV 文件放到服务器上，并且让它处在 MySQL 能读取的一个目录下。为了安全起见，这个目录应该只有 MySQL 才能读取。不过，为了方便，这次我们就把它临时放到 /tmp 目录下。

接下来，建一个用于接收这些数据的表。虽然这个 CSV 文件含有的行和列超过了我们的需要，但导入 32 000 行也不过是几秒钟的事。所以，不用太担心大小问题。

尽管可以把它直接导入现存的表，但最好还是建一个专门用于导入数据的表。这样，当要把数据复制到现存的表中时，用 INSERT INTO...SELECT 即可。建表语句如下：

```
CREATE TABLE rookery.clements_list_import
(id INT, change_type VARCHAR(255),
col2 CHAR(0), col3 CHAR(0),
scientific_name VARCHAR(255),
english_name VARCHAR(255),
col6 CHAR(0), `order` VARCHAR(255),
family VARCHAR(255),
col9 CHAR(0), col10 CHAR(0),
col11 CHAR(0), col12 CHAR(0),
col13 CHAR(0), col14 CHAR(0),
col15 CHAR(0), col16 CHAR(0), col17 CHAR(0));
```

这个表的列会与 CSV 文件的每个域一一对应，而且顺序也一样。对于那些用不到的域，我们就直接用号数来给它们起名，并把类型设为 CHAR(0)（定长字符串，且长度为零），这样这些域就不会被存储了。其实还有一个更好的做法，就是只导入我们想要的列。我们在本章后面会讲到这种做法。暂时就用简单的方法，你只要忽略多出的列即可。

那些有用的列则被赋予了与原域意义接近的名字，并设为 VARCHAR(255) 型。注意，order 列需要加反引号，因为 order 本身是保留字（比如，用于 ORDER BY）。只要在用到该列时都带上反引号，那么命名为 order 是没有问题的。否则，MySQL 就无法辨认它，并导致出现错误。

至此，我们已下载好用来导入的数据文件，也已将它放到了 MySQL 能查看的目录下。我们决定了该文件的格式，并按该格式创建了一个表来接收数据。下面就可以进行数据导入了。

15.2　导入数据的基本做法

要把数据导入 MySQL 或 MariaDB，需要使用带有 FILE 权限的管理员账号，而第 13 章所建的 admin_import 正符合要求。

我们可以用 LOAD DATA INFILE 来从文本文件中加载数据。它是多功能的语句，可带各种选项和子句。本章会对它们逐一进行讲解。如果你想以最简单的方式把 Clements-Checklist-6.9-final.csv 的数据导入 clements_list_import 表，可输入以下命令：

```
LOAD DATA INFILE '/tmp/Clements-Checklist-6.9-final.csv'
INTO TABLE rookery.clements_list_import
FIELDS TERMINATED BY ',';
```

首先，我们注意到，在这条 SQL 语句中，文件路径与文件名位于引号之间。而到底使用单引号还是双引号，是无所谓的。然后，看看 FIELDS 子句。该子句告诉 MySQL，文件中的各个域是如何分隔的。因为我们导入的文件是以逗号来划分域的，所以 FIELDS 子句加了 TERMINATED BY 嵌套子句，以及一个逗号（需加引号）。

虽然还有其他子句和嵌套子句，但这就是 LOAD DATA INFILE 的基本写法。不过这样写其实是有点问题的，它会生成警告信息。

15.2.1　检查警告信息

输入上例的语句，你会看到很多警告信息。以下是执行该语句后的返回信息，以及 SHOW WARNINGS 的头几行：

```
Query OK, 32187 rows affected, 65535 warnings (0.67 sec)
Records: 32187 Deleted: 0 Skipped: 0 Warnings: 209249

SHOW WARNINGS;

+---------+------+-------------------------------------------------------------+
| Level   | Code | Message                                                     |
+---------+------+-------------------------------------------------------------+
| Warning | 1366 | Incorrect integer value: 'sort 6.9' for column 'id' at row 1 |
| Warning | 1265 | Data truncated for column 'col2' at row 1                   |
| Warning | 1265 | Data truncated for column 'col3' at row 1                   |
...
```

可以执行 SHOW WARNINGS 语句，查看所有警告信息。而因为刚才总共产生了 209 249 个警告，所以我就只列举了开头部分。事实上，剩下的那些警告说的都是类似的问题。这些问题中出现最多的是说，你把数据文件中有值的域导入了长度为零的 CHAR 列。遇到这种情况时，MySQL 就会自动将值截短，而后再塞到列中，并报告每个进行过这种操作的列。我们可以研究一下表中的数据样本，以便更清楚地看出这到底是什么问题，以及 MySQL 做了怎样的处理：

```
SELECT * FROM rookery.clements_list_import LIMIT 2 \G;

*************************** 1. row ***************************
            id: 0
```

```
    change_type: Clements 6.9 change
           col2:
           col3:
scientific_name: Scientific name
   english_name: English name
           col6:
          order: Order
         family: Family
           col9:
          col10:
          col12:
          col13:
          col14:
          col15:
          col16:
          col17:
*************************** 2. row ***************************
             id: 1
    change_type:
           col2:
           col3:
scientific_name: Struthio camelus
   english_name: Ostrich
           col6:
          order: Struthioniformes
         family: Struthionidae (Ostrich)
           col9:
          col10:
          col12:
          col13:
          col14:
          col15:
          col16:
          col17:
```

这样看来，我们的 LOAD DATA INFILE 操作是成功的。数据文件的域和表的列都对上了。不过，第一行导入了数据文件的表头。虽然我们不需要它，但这也没什么大碍。第二行就清楚地显示出，我们把数据都导入了正确的列：鸟的学名、俗名、目名、科名都齐了。而那些我们不需要的域，其对应的列也确实没导入任何数据。这样是没什么问题的。不过，我之前也说过，其实我们可以把表做得更整洁，使我们显得更专业。具体的做法稍后再谈。先把新鸟种加入 birds 表。

15.2.2　检查导入是否准确

在往 birds 表中插入数据之前，先检查 clements_list_import 表中的数据导入得是否正确。输入以下 SELECT 语句，查出 new species 的行，并检查结果：

```
SELECT id, change_type,
scientific_name, english_name,
`order`, family
FROM rookery.clements_list_import
WHERE change_type = 'new species' LIMIT 2 \G
```

```
*************************** 1. row ***************************
             id: 4073
    change_type: new species
scientific_name: species
   english_name: Prosobonia ellisi
          order: extinct; formerly Moorea (Society Islands)
         family: Charadriiformes
*************************** 2. row ***************************
             id: 6707
    change_type: new species
scientific_name: from the Arabian Peninsula
   english_name: with range ""central Al Hajar mountains
          order: species
         family: Strix omanensis
```

我只查了两行，不过你可以去掉 LIMIT 子句，查出所有行。总共应该是 11 行。这里显示的两行就来自我之前截取 Clements-Checklist-6.9-final.csv 的部分。而从这个 SELECT 可以发现，数据导入得不太对。为了看得更清楚，我们回到文件中，以第二行为例：

```
6707,
new species,
"Robb et al. (2013) describe a new species of owl,
 Omani Owl (Strix omanensis),
 from the Arabian Peninsula,
 with range ""central Al Hajar mountains,
 northern Oman"". Position Omani Owl immediately following
 Hume's Owl (Strix butleri).",
species,
Strix omanensis,
Omani Owl,
"central Al Hajar mountains,
northern Oman",
Strigiformes,
Strigidae (Owls),
,,,,addition (2014),,,,,
```

这里，我把插入列中的文本显示成粗体了。这样看来，似乎是 MySQL 被域中的逗号迷惑了。因为我们指定了 FIELDS TERMINATED BY ','，所以就导致了含逗号的域被拆分，并被填到后续的列中。要想解决这个问题，可以在 FIELDS 子句中增加一些参数来判断。

让我们先清空 clements_list_import 表，然后重新导入一次。这就是创建临时表的好处之一，我们可以随时清空，从头再来。使用以下两条语句来做：

```
DELETE FROM rookery.clements_list_import;

LOAD DATA INFILE '/tmp/Clements-Checklist-6.9-final.csv'
INTO TABLE rookery.clements_list_import
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
IGNORE 1 LINES;
```

第一句是清表，以便重导。而第二句与之前的 LOAD DATA INFILE 差不多，不过给 FIELDS 子句加了 ENCLOSED BY 嵌套子句，来指定域用的是双引号。另外，因为并非所有域都这样，

所以我们还要在这个嵌套子句前加上 OPTIONALLY 选项。这样就使得 MySQL 在遇到一个双引号时，会往后寻找另一个，并在找到时，把这一对双引号之间的内容都当成数据（即使其中含有逗号）。

由于有些域不带双引号，而双引号又不止一对，你可能会觉得应该很难分辨。但对于 MySQL 来说，这根本不成问题。

> 一般来说，正在导入数据的表是被锁定的，其他用户不能操作它。但其实，可以带上 LOW_PRIORITY 选项，使别人能读取该表，即使用 LOAD DATA LOW_PRIORITY INFILE。这样，导入数据的行为就会等到没人读表的时候才被执行。此选项只在具有表级锁的表（存储引擎为 MyISAM）上生效，而在行级锁的表（存储引擎为 InnoDB）上无效。

此外，我还在末尾加了 IGNORE 子句，令 MySQL 忽略我所指定的行数（从文件开头算起）。因此，第一行的表头就不会被导入数据库了。如果表头不止一行，可以在 IGNORE 后指定相应的数字。

再执行一次之前的 SELECT：

```
SELECT id, change_type,
scientific_name, english_name,
`order`, family
FROM rookery.clements_list_import
WHERE change_type = 'new species' LIMIT 2 \G

*************************** 1. row ***************************
             id: 4073
    change_type: new species
scientific_name: Prosobonia ellisi
   english_name: Moorea Sandpiper
          order: Charadriiformes
         family: Scolopacidae (Sandpipers and Allies)
*************************** 2. row ***************************
             id: 6707
    change_type: new species
scientific_name: Strix omanensis
   english_name: Omani Owl
          order: Strigiformes
         family: Strigidae (Owls)
```

这次导入正确了，学名和俗名所在的列，以及我们需要的其他列都对上了。现在可以进行下一步了。

15.2.3 选取导入的数据

既然我们已经把康奈尔大学的数据文本正确地导入 clements_list_import 表，那就可以用 INSERT INTO...SELECT 将该表的数据复制到 birds 表中了。因为我们现在还在学习和实验阶段，所以先来创建一个结构如同 birds 的表，容纳来自 clements_list_import 的数据。建表语句如下：

```
CREATE TABLE rookery.birds_new
LIKE rookery.birds;
```

接着，从 clements_list_import 表中选取我们需要的行，插入 birds_new 表。在服务器上执行以下命令：

```
INSERT INTO birds_new
   (scientific_name, common_name, family_id)
SELECT clements.scientific_name, english_name, bird_families.family_id
  FROM clements_list_import AS clements
  JOIN bird_families
     ON bird_families.scientific_name =
     SUBSTRING(family, 1, LOCATE(' (', family) )
  WHERE change_type = 'new species';
```

这里只取了 clements_list_import 表的两列来插入（即 scientific_name 和 english_name）。而 family_id 则通过将 clements_list_import 与 bird_families 连接而查得。为了确定 family_id，我们需要用科名来做连接。虽然 clements_list_import 表中有 family 列，但它的科名中还包含了多余的文字（该科中一些鸟的俗名）。所以，需要用 LOCATE() 来查出该科名中空格后接一个左括号的位置，然后用 SUBSTRING() 把该位置前的文字截取出来当作科名（例如，从 Strigidae (Owls) 中截出 Strigidae）。而为了筛选出新鸟种，我们在 WHERE 中限定只查出 change_type 为 new species 的行。

来看看这个 INSERT INTO...SELECT 效果如何：

```
SELECT birds_new.scientific_name,
common_name, family_id,
bird_families.scientific_name AS family
FROM birds_new
JOIN bird_families USING(family_id);

+-----------------------+--------------------+-----------+---------------+
| scientific_name       | common_name        | family_id | family        |
+-----------------------+--------------------+-----------+---------------+
| Prosobonia ellisi     | Moorea Sandpiper   |       164 | Scolopacidae  |
| Strix omanensis       | Omani Owl          |       178 | Strigidae     |
| Batrachostomus chaseni| Palawan Frogmouth  |       180 | Podargidae    |
| Erythropitta yairocho | Sulu Pitta         |       217 | Pittidae      |
| Cichlocolaptes maza...| Cryptic Treehunter |       223 | Furnariidae   |
| Pomarea nukuhivae     | Nuku Hiva Monarch  |       262 | Monarchidae   |
| Pomarea mira          | Ua Pou Monarch     |       262 | Monarchidae   |
| Pnoepyga mutica       | Chinese Cupwing    |       285 | Pnoepygidae   |
| Robsonius thompsoni   | Sierra Madre Gro...|       290 | Locustellidae |
| Zoothera atrigena     | Bougainville Thrush|       303 | Turdidae      |
| Sporophila beltoni    | Tropeiro Seedeater |       322 | Thraupidae    |
+-----------------------+--------------------+-----------+---------------+
```

看起来不错。11 行新鸟种都在了，而且科名也对应。接下来只需再用 INSERT INTO...SELECT 把 birds_new 表的数据复制到 birds 表中即可。

尽管这个 CSV 文件中有大量的数据，但导入起来并不困难。如果你想导入得更加顺畅，可以在不同的情况中，使用其他不同的子句、嵌套子句和选项。接下来的几节就来介绍它们。

15.3 更好地导入

尽管我们已经成功地导入了一个大文件，但其实可以做得更好。所以，本节将会介绍如何更好地使用 LOAD DATA INFILE。

15.3.1 对应域

在之前的导入中，为了排除那些我们不想要的域，我们给这些域创建了长度为零的字符型列，并且，对于由此而产生的大量警告，我们选择了不去理会。

但其实，有更好的方法来过滤掉那些不想要的域。可以在 LOAD DATA INFILE 的末尾，以逗号分隔的方式，给出一些列名或用户自定义变量。这些列名和变量的数量必须与文件中域的数量一致，而且，列名应该与其该存储的域对应起来，按文件中域的排序来写。即使该排序与表结构的排序不一样，也无所谓，即表结构不必与文件一致。而对于不想要的域，可以用临时变量来对应，并任由这些变量的值在导入过程中被反复改写。最后，你也不必理会它们，因为临时变量会在连接断开时被清除。

现在，先删除 clements_list_import 表，并重建一个不带那些无用列的表。顺便修改列的顺序。做法如下：

```
DROP TABLE rookery.clements_list_import;

CREATE TABLE rookery.clements_list_import
(id INT, scientific_name VARCHAR(255),
english_name VARCHAR(255), family VARCHAR(255),
bird_order VARCHAR(255), change_type VARCHAR(255));
```

于是，这个导入数据的表就只包含我们想要的列了。其中，family 在 bird_order 之前，而最后是 change_type。

接着，再次导入数据。这次带上一堆列名和变量，分别与想要的和不想要的域对应起来。那些不想要的域会被导入临时变量 @niente。niente 在意大利语中是“无”的意思。其实变量的命名可随意。现在，执行以下 SQL 语句：

```
LOAD DATA INFILE '/tmp/Clements-Checklist-6.9-final.csv'
INTO TABLE rookery.clements_list_import
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
IGNORE 1 LINES
(id, change_type, @niente, @niente,
scientific_name, english_name,
@niente, bird_order, family, @niente,
@niente, @niente, @niente, @niente,
@niente, @niente, @niente, @niente);

Query OK, 32180 rows affected (0.66 sec)
Records: 32180 Deleted: 0 Skipped: 0 Warnings: 0
```

注意，列名和变量的排序需要与 CSV 文件的域对应。即使最后顺序与表结构不一样，MySQL 也会帮你处理好。而那些不想要的域则被导入了临时变量 @niente，这使得每次读到下一个域时，该变量的值都会改变。导入过程很顺利，没有任何警告。现在，让我们选

取表中最后的两个新鸟种，看看结果如何：

```
SELECT * FROM rookery.clements_list_import
WHERE change_type='new species'
ORDER BY id DESC LIMIT 2 \G

*************************** 1. row ***************************
             id: 30193
scientific_name: Sporophila beltoni
   english_name: Tropeiro Seedeater
         family: Thraupidae (Tanagers and Allies)
     bird_order: Passeriformes
    change_type: new species
*************************** 2. row ***************************
             id: 26879
scientific_name: Zoothera atrigena
   english_name: Bougainville Thrush
         family: Turdidae (Thrushes and Allies)
     bird_order: Passeriformes
    change_type: new species
```

如果你用的文件与我的不同，那么可能不是这样的结果。无论如何，从这里可以发现，数据能被导入正确的列。接下来，只要运行 INSERT INTO...SELECT，就能将新鸟种复制到 birds_new 表中，再到 birds 表中。如果你对自己输入数据的能力有信心，也可以直接复制到 birds 表中。现在这种做法已经比上次好多了，但仍有改进空间。下面我们再导一次，不过这次要去掉 family 列中的俗名。

15.3.2 设置列

如果你想在把数据导入表的列之前，先处理数据，可以在 LOAD DATA INFILE 中用 SET 子句。在之前的例子中，我们在 INSERT INTO...SELECT 中用了 SUBSTRING() 来去除 clements_list_import 表的 family 列中的描述（用括号括起来的俗名）。下面我们尝试在导入数据时清除描述。用下面这两条语句来删除并重导数据：

```
DELETE FROM rookery.clements_list_import;

LOAD DATA INFILE '/tmp/Clements-Checklist-6.9-final.csv'
INTO TABLE rookery.clements_list_import
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
IGNORE 1 LINES
(id, change_type, @niente, @niente,
scientific_name, english_name,
@niente, bird_order, @family, @niente,
@niente, @niente, @niente, @niente,
@niente, @niente, @niente, @niente, @niente)
SET family = SUBSTRING(@family, 1, LOCATE(' (', @family) );
```

这与之前的 LOAD DATA INFILE 差不多，但这次我们先把科名存到了 @family 变量中，然后再用 SUBSTRING() 抽取 @family 中的子字符串，最后用 SET 子句将该返回值存入 family 列。可以用 Treehunter 这个新鸟种为例，看看结果如何：

```
SELECT * FROM rookery.clements_list_import
WHERE change_type='new species'
AND english_name LIKE '%Treehunter%' \G

*************************** 1. row ***************************
             id: 13864
scientific_name: Cichlocolaptes mazarbarnetti
   english_name: Cryptic Treehunter
         family: Furnariidae
     bird_order: Passeriformes
    change_type: new species
```

很明显，数据都在正确的列中，而且，family 中鸟的俗名没有了。如果需要的话，你现在就可以再次用 INSERT INTO...SELECT 将新鸟种的数据复制到 birds 表中。

15.4 其他格式的域和行

数据文件的结构可能多种多样，并不一定都如上例中使用的康奈尔大学的 CSV 文件那样。有一些文件的域和行的格式不一样。所以，我们还应导入一个不同的数据文件，学习一下用 LOAD DATA INFILE 语句来定义各种域和行的其他方法。

接下来的例子有点像 10.2 节中的例子。当时，营销人员给我们的是一个 dump 文件，而这次，假设他们给的是一个叫 birdwatcher-prospects.csv 的数据文件。它包含 Rookery 网站的潜在用户的姓名和邮件地址。你可以从 MySQL 资源站（http://mysqlresources.com/files）下载这个文件的副本。该文件的头几行如下：

```
["prospect name"|"prospect email"|"prospect country"]
["Mr. Bogdan Kecman"|"bodgan\@kecman-birds.com"|"Serbia"]
["Ms. Sveta Smirnova"|"bettasveta\@gmail.com"|"Russia"]
["Mr. Collin Charles"|"callincollin\@gmail.com"|"Malaysia"]
["Ms. Sveta A. Smirnova"|"bettasveta\@gmail.com"|"Russia"]
```

域的名字在第一行。每行都以左中括号开头，并以右中括号结尾。每个域都有双引号包围，而域之间还有竖线分隔。@ 前的反斜杠是跳脱符号，用于说明其后的字符应按字面意思来理解。若要导入这样的文件，需要告知 MySQL 应如何识别行头、行尾，以及跳脱符号。

15.4.1 开始、结束和跳脱

导入 birdwatcher-prospects.csv 文件前，先建好容纳数据的表。该表除了要有与原文件三域对应的三列，还应加一个自增列作为主键。而因为邮件地址通常是每个人独有的，所以该列应设为 UNIQUE。建表语句如下：

```
CREATE TABLE birdwatchers.birdwatcher_prospects_import
(prospect_id INT AUTO_INCREMENT KEY,
prospect_name VARCHAR(255),
prospect_email VARCHAR(255) UNIQUE,
prospect_country VARCHAR(255));
```

建好之后，就可以从 birdwatcher-prospects.csv 导入数据了。执行以下语句：

```
LOAD DATA INFILE '/tmp/birdwatcher-prospects.csv'
INTO TABLE birdwatchers.birdwatcher_prospects_import
FIELDS TERMINATED BY '|' ENCLOSED BY '"' ESCAPED BY '\\'
LINES STARTING BY '[' TERMINATED BY ']\r\n'
IGNORE 1 LINES
(prospect_name, prospect_email, prospect_country);
```

虽然以上语句是正确的，但如果加载 birdwatcher-prospects.csv，它就会报错，导入不了数据。我们会在下一小节处理这个错误。现在，先关注一下 `LOAD DATA INFILE` 中 `FIELDS` 和 `LINES` 子句的嵌套子句。

首先，看看 `FIELDS` 子句。

- `TERMINATED BY` 的意思是，域是以竖线结尾的。虽然最后一个域的后面没有竖线，但因为我们指定了行的结束符号，所以 MySQL 也能确定每行最后一个域在哪里结束。
- `ENCLOSED BY` 的意思是，每个域都被放在双引号中。
- `ESCAPED BY` 用于指定跳脱符号。其实默认就是反斜杠，所以对于这份文件，我们不需要带这个嵌套子句。我写出来只是想让你知道有这回事。

接着，看看 `LINES` 子句。

- `STARTING BY` 的意思是，行以左中括号开头。
- `TERMINATED BY` 指定了行由右中括号、回车和换行符结尾。通常，回车是不需要的，但因为这份文件是由 Windows 上以这种方式结尾的软件所产生的，所以我们得这么写。

15.4.2 替换数据或忽略错误

现在，我们来处理上一小节中 `LOAD DATA INFILE` 所报的错。在我们运行 SQL 语句时出现了如下报错信息：

```
ERROR 1062: Duplicate entry 'bettasveta@gmail.com' for key 'prospect_email'
```

报错的原因是，文件中有两行邮件地址相同（都是 Sveta Smirnova 的邮件地址），但 `prospect_email` 列已被设为 `UNIQUE`。于是，这就导致了整个导入都被回滚，最终没有数据能进入表中。

对于这种问题，有几种处理方法。例如，我们可以修改 `prospect_email` 列的 `UNIQUE` 设置，从而允许邮件地址重复。也可以让 MySQL 忽略这样的错误。要做到这一点，可以给 `LOAD DATA INFILE` 加上 `IGNORE` 选项，如下所示：

```
LOAD DATA INFILE '/tmp/birdwatcher-prospects.csv'
IGNORE INTO TABLE birdwatchers.birdwatcher_prospects_import
FIELDS TERMINATED BY '|' ENCLOSED BY '"' ESCAPED BY '\\'
LINES STARTING BY '[' TERMINATED BY ']\r\n'
IGNORE 1 LINES
(prospect_name, prospect_email, prospect_country);

Query OK, 4 rows affected, 1 warning (0.02 sec)
```

```
Records: 5  Deleted: 0  Skipped: 1  Warnings: 1

SHOW WARNINGS \G

*************************** 1. row ***************************
  Level: Warning
   Code: 1062
Message: Duplicate entry 'bettasveta@gmail.com' for key 'prospect_email'
```

运行起来很顺利。注意返回信息，它说有一行被跳过了，还产生了一个警告。而这个警告则是说，有重复记录。该重复记录也正是被跳过的那行，即 Sveta Smirnova 的第二条记录。现在用 SELECT 来看看 Sveta Smirnova 的那行：

```
SELECT * FROM birdwatchers.birdwatcher_prospects_import
WHERE prospect_name LIKE '%Sveta%' \G

*************************** 1. row ***************************
     prospect_id: 16
   prospect_name: Ms. Sveta Smirnova
  prospect_email: bettasveta@gmail.com
prospect_country: Russia
```

结果显示，文件中 Sveta Smirnova 的第一条记录被插入表中了，而第二条则没有，因为第二条的名字带有中间名首字母，很明显这里不是。如果你想插入第二条，那么可以把 IGNORE 换成 REPLACE：

```
LOAD DATA INFILE '/tmp/birdwatcher-prospects.csv'
REPLACE INTO TABLE birdwatchers.birdwatcher_prospects_import
FIELDS TERMINATED BY '|' ENCLOSED BY '"' ESCAPED BY '\\'
LINES STARTING BY '[' TERMINATED BY ']\n'
IGNORE 1 LINES
(prospect_name, prospect_email, prospect_country);

Query OK, 6 rows affected (0.02 sec)
Records: 5 Deleted: 1 Skipped: 0 Warnings: 0

SELECT * FROM birdwatchers.birdwatcher_prospects_import
WHERE prospect_name LIKE '%Sveta%' \G

*************************** 1. row ***************************
     prospect_id: 26
   prospect_name: Ms. Sveta A. Smirnova
  prospect_email: bettasveta@gmail.com
prospect_country: Russia
```

返回信息说跳过的有 0 行，删除的有 1 行。这是因为 Sveta Smirnova 的第二条记录把第一条替换了。从 SELECT 的结果可以看出，现在只留下了带有中间名首字母的那条记录。

15.5　在MySQL之外导入数据

至今我们介绍的导入方法都是在 MySQL 中操作的。其实，不登录 MySQL 也可以导入数据。至少，你可以用 mysql 的 --execute 选项执行 LOAD DATA INFILE，也可以用另一个专

门导入数据的工具，即 mysqlimport。本节就来介绍它。与 LOAD DATA INFILE 一样，它需要账号有 FILE 权限。但是如果你没有 FILE 权限，也有其他方法。现在，我们先谈谈如何不上传到服务器，而直接导入本地文件。

15.5.1 导入本地文件

即使你没有权限将文件上传到服务器，也可以直接通过 mysql 来导入数据。具体做法就是加上 LOCAL 选项。无需先登录服务器再以 localhost 的方式开启 mysql 客户端，相反，在本地计算机上用类似以下的命令来进行本地登录：

```
mysql --user=admin_import --password \
      --host=mysqlresources.com --database=rookery
```

通过本地客户端建立好连接后，就可以用如下语句来导入了：

```
LOAD DATA LOCAL INFILE '/tmp/birdwatcher-prospects.csv'
REPLACE INTO TABLE birdwatchers.birdwatcher_prospects_import
FIELDS TERMINATED BY '|' ENCLOSED BY '"' ESCAPED BY '\\'
LINES STARTING BY '[' TERMINATED BY ']\n'
IGNORE 1 LINES
(prospect_name, prospect_email, prospect_country);
```

基本上，这会使客户端读取该文件，并发送给服务器，而服务器则会将这些内容暂存在操作系统的临时目录中（比如 /tmp）。

这种做法需要服务器和客户端都接受 LOCAL 选项，即需要有人在两边的配置文件中都设置 local-infile=1。另外，登录的用户账号必须要有 FILE 权限。虽然它通常不会被授予一般的用户，但如果那个服务器是你的，那么大可以这样授权。授权问题可参考第 13 章。

15.5.2 使用mysqlimport

如果你收到的数据文件都有同样的格式，那么写一个简单的 shell 脚本，让它自动将数据导入 MySQL 应该会很方便。对于这种自动导入的任务，可以使用 mysqlimport 来做。它会按指定的选项来执行 LOAD DATA INFILE。

以之前的 birdwatcher-prospects.csv 为例，来演示这个工具的用法。mysqlimport 要求文件名和表名一致，所以，我们要把文件改名为 birdwatcher_prospects.csv。稍后我会再解释为什么。现在，先试着在服务器的命令行中执行以下命令：

```
mysqlimport –user='marie_dyer' --password='sevenangels' \
  --replace --low-priority --ignore-lines='1' \
  --fields-enclosed-by='"' --fields-terminated-by='|' --fields-escaped-by='\\' \
  --lines-terminated-by=']\r\n' \
  --columns='prospect_name, prospect_email, prospect_country' \
    birdwatchers '/tmp/birdwatcher_prospects_import.csv'
```

如你所见，所有的选项与 LOAD DATA INFILE 的一样，只是变成了小写，并以两个横杠开头。选项的顺序是没有规定的，但数据库名和文件名要放在命令的最后。文件名可以有多个，用空格将它们分开，然后 mysqlimport 会按照你给的顺序来导入它们。

文件名的前缀必须与表名一样，句点及文件扩展名不算。`mysqlimport` 就是这样依据文件名来决定将数据导入哪个表的。因为表名不能带有横杠（MySQL 可能会将横杠当成减号），所以文件名也要相应地将横杠改成下划线。

`mysqlimport` 的运作就如同 `LOAD DATA INFILE`，而事实上，它在内部调用的正是 `LOAD DATA INFILE`。有了这个工具，你就可以将导入命令写在 shell 脚本中，或者让命令作为 `crontab` 的一条，使 `mysqlimport` 能定期自动导入你不断更替的数据文件。

> 你可能已发现，上例中没有 `--lines-starting-by` 选项。那是因为 `mysqlimport` 确实没有这个选项。专门研究 MySQL 软件的著名专家 Paul Dubois 在 2006 年就提出过这个问题。但这个选项至今仍然没有被添加。这也表明，`mysqlimport` 的支持很乏力。事实上，在服务器上测试它时，我发现它不太好用。如果它适用于你的环境，那当然很好。如果你要写脚本来导入数据，不妨使用 API 脚本（见第 16 章）再配合 `LOAD DATA INFILE` 来做。而且，大多数脚本语言都能做各种文件格式的转换。

15.5.3　没有FILE权限也能导入数据

因为担心安全问题，所以有些主机托管公司会通过不授予 `FILE` 权限，令你无法使用 `LOAD DATA INFILE`。但其实你可以绕过它，只是必须要完成一些额外的步骤。

首先，登录你有 `FILE` 权限的另一个 MySQL 服务器（可以是你的个人计算机上的 MySQL）。我们把这个 MySQL 叫作*临时环境*，而把最终要导入数据的 MySQL 叫作*正式环境*。在临时环境中，建一个与正式环境中用于导入数据的表一模一样的表。而在正式环境中，最好也与之前的例子一样，建一个暂存数据的表，而不是直接把数据导入最终的表。

两边都建好后，在临时环境中执行 `LOAD DATA INFILE` 导入数据。

接着，用第 14 章详细介绍过的 `mysqldump` 在临时环境中把该表的数据导出。记得加上 `--tables` 选项，以确保只导出需要导入数据的表（可以参考 14.1.5 节），并加上 `--no-create-info` 选项，使 dump 文件不含有 `CREATE DATABASE` 和 `CREATE TABLE`。

创建表的 dump 文件后，把它上传到正式环境。在正式环境中，用 `mysql` 将这个 dump 文件导入需要导入文件的表（具体做法可以参考 14.2 节）。最后，用 `INSERT INTO...SELECT` 将数据复制到相应的表中。

其实这种导入数据的做法与之前的差不多，只是多了这样几个步骤：在临时环境中加载数据，并用 `mysqldump` 导出 dump 文件，再在正式环境中用 `mysql` 将 dump 文件导入相应的表。这种做法并不困难，只是比较费时而已。

15.6　批量导出数据

到现在为止，本章讲的都是如何把文本文件里的数据批量导入 MySQL 和 MariaDB。但某些时候，可能会有人要你反过来做：从 MySQL 数据库中将数据批量导出到文本文件中。

其实与导入相比，导出简单得多，最多就是要想好导出成什么格式。

批量导出数据的最简单的方法，就是用 SELECT 并带上 INTO OUTFILE 子句。它运行起来与 LOAD DATA INFILE 差不多，因为用到的嵌套子句都是一样的，只是一个导出，一个导入。下面我们就来看个例子。

假设现在我们要提供 rookery 中 Charadriiformes 目（包含 Sea Gull 和 Plover）的鸟种列表。我们要导出鸟种的学名、俗名，以及科名。

先在临时环境中试试。首先，写一条 SELECT 语句，确保导出的数据符合要求：

```
SELECT birds.scientific_name,
IFNULL(common_name, ''),
bird_families.scientific_name
FROM rookery.birds
JOIN rookery.bird_families USING(family_id)
JOIN rookery.bird_orders USING(order_id)
WHERE bird_orders.scientific_name = 'Charadriiformes'
ORDER BY common_name;
```

因为要获取特定目的鸟种的名字及其科名，所以我们在 SELECT 中用了 JOIN（见第 9 章）来连接 rookery 中那三个主要的表。最后，我们还加上了 ORDER BY 来按 common_name 排序。而因为 SELECT...INTO OUTFILE 通常会将 NULL 值转换成字母 N，所以我们还要用 IFNULL() 将所有 NULL 的俗名转换成空字符串。这个 SELECT 是没问题的。如果你在服务器上试试，会发现它大概会返回 718 行。

为了让接收文件的人能看懂每个域代表什么意思，我们要在第一行加上每个域的名字。最简单的做法就是在 SELECT 后紧跟一堆字符串：

```
SELECT 'scientific name','common name','family name';
```

这些名字不需要与列名一样，也不需要遵循特别的规则。为了把以上语句加到导出结果里，我们用 UNION 来拼接两个 SELECT（注意，域名字的 SELECT 要打头）。UNION 的用法也在第 9 章讲过。

测试过这些 SELECT 之后，就可以结合它们来导出数据了。执行以下命令：

```
( SELECT 'scientific name','common name','family name' )
UNION
( SELECT birds.scientific_name,
  IFNULL(common_name, ''),
  bird_families.scientific_name
  FROM rookery.birds
  JOIN rookery.bird_families USING(family_id)
  JOIN rookery.bird_orders USING(order_id)
  WHERE bird_orders.scientific_name = 'Charadriiformes'
  ORDER BY common_name
  INTO OUTFILE '/tmp/birds-list.csv'
  FIELDS ENCLOSED BY '"' TERMINATED BY '|' ESCAPED BY '\\'
  LINES TERMINATED BY '\n');
```

导出应该是能成功的。讲过 SELECT 的部分后，我们来关注第二个 SELECT 中的 INTO

OUTFILE 子句。首先要注意，我们指定了导出的路径是 /tmp。MySQL 一般只能往它能访问的目录中写数据，而 /tmp 就是服务器上所有用户都能读写的目录之一。还要注意，这里的嵌套子句是放在文件路径和文件名后面的，这和 LOAD DATA INFILE 相反。不过，它们用到的嵌套子句都是一样的。

我们指定了域要用双引号包围，且以竖线分隔，并设置了反斜杠作为跳脱符号。因为 SELECT...INTO OUTFILE 没有默认的跳脱符号设置，所以你得自己加上 ESCAPED BY。而这里之所以要写两个反斜杠，是因为反斜杠本身也要跳脱。最后，设置 \n 作为每行的结尾。

导出的文件大概如下（我只截取了开头几行）：

```
"scientific name"|"common name"|"family name"
"Charadrius vociferus"|"Killdeer"|"Charadriidae"
"Charadrius montanus"|"Mountain Plover"|"Charadriidae"
"Charadrius alexandrinus"|"Snowy Plover"|"Charadriidae"
"Pluvialis squatarola"|"Black-bellied Plover"|"Charadriidae"
"Pluvialis fulva"|"Pacific Golden Plover"|"Charadriidae"
"Burhinus vermiculatus"|"Water Thick-knee"|"Burhinidae"
"Burhinus oedicnemus"|"Eurasian Thick-knee"|"Burhinidae"
...
```

看上去不错。第一行是域的名字。接下来是整理好格式的数据。这样十分有利于给使用另一种数据库系统的人提供数据。

15.7　小结

尽管你可能很少用得上 LOAD DATA INFILE，但当你需要使用它时，会发现它能帮你节约很多时间。它使批量导入数据和迁移到 MySQL 和 MariaDB 变得轻松简单。虽然数据文件格式多样，你可能需要尝试好几次才能准确导入数据，但只要你是在临时的表上操作，就可以反复导入和删除数据，而不影响别人，也不会有丢失数据的风险。

而 SELECT...INTO OUTFILE 则是分享数据的绝佳工具。如果你所在的公司有向其他公司分享数据的习惯，那么它应该会成为你的常用工具。所以，你至少应该熟悉这个工具，以备不时之需。

15.8　习题

要完成本章的习题，需要先从 MySQL 资源站（http://mysqlresources.com/files）下载 employees.csv 和 birder-list.csv，并把它们复制到 /tmp 或其他 MySQL 能读写的目录里。

其中，employees.csv 是我用 SELECT...INTO OUTFILE 把 employee 数据库导出而生成的。而这个巨大的样本数据库则是由 MySQL 的人员创建的，可供免费下载（https://launchpad.net/test-db/）。

(1) 用文本编辑器打开 employees.csv，看看它是怎样的格式。然后创建一个与之匹配的需要导入数据的表。建好后，用 LOAD DATA INFILE 把 CSV 文件中的数据导进表里。

(2) 用文本编辑器打开 birder-list.csv，看看它是怎样的格式。你会发现它包含一些意大利人，而且这些人有望成为我们网站的会员。所以，在 birdwatchers 库中，创建一个需要导入数据的表，令其拥有如下顺序的列：id、formal_title、name_first、name_last、country 和 email。将 id 列设为自增的键。

构造一条 LOAD DATA INFILE 语句，将 birder-list.csv 的数据导入刚才那个表。记得要指定列名。formal_title 列的填写，需要用到 SET 子句。因为意大利的女性名字通常以 a 结尾，而男性名字通常以 o 结尾（也有一些以 e 或 i 结尾），所以，可根据此规律来让 MySQL 在 formal_title 中填 Ms. 或 Mr.。写完后，执行 LOAD DATA INFILE 导入数据。

导入完成后，用 SELECT 检查数据导入得是否正确。如果不正确，就清空表里的数据，然后重新导入，直到正确为止。只要成功导入了，就用 INSERT INTO...SELECT 将这些名字加到 humans 表中。

(3) 用 SELECT...INTO OUTFILE 把 birds 表中俗名带有 Least 的鸟导出到一个叫 little-birds.csv 的文件中。导出的结果需要包含鸟种的俗名、学名，以及所属科、目的学名。格式如下：域要用双引号包围，域之间用逗号分隔，每条记录以分号结尾，不要换行（即不带 \n 或 \r）。如无意外，这会使 CSV 文件把所有内容写成很长的一行。导出数据后，用文本编辑器打开该文件，确认数据是否在一行中。

(4) 在 rookery 数据库中，创建一个叫作 birds_least 的表。其中包含四列：scientific_name、common_name、family_name 和 order_name。然后，用 LOAD DATA INFILE 把上题生成的 little-birds.csv 导入此表。这可能需要一点技巧。如果导入不成功，那就删除表中的数据，重新导入，直到成功为止。

第 16 章

应用编程接口

所谓使用应用编程接口（Application Programming Interface，API），就是使用编程语言与计算机软件系统进行交互。它的优点是能让你按自己的需要，打造个性化的用户界面。大型网站都使用 API 来让外界与 MySQL 和 MariaDB 数据库进行交互，这样用户便不需要了解他们正在使用的数据库，也不需要懂 SQL 语句。

本章会介绍几种用于与 MySQL 和 MariaDB 交互的 API，以便令你能编写自定义应用程序来操作数据库。内容包括：C API、Perl DBI、PHP API、Connector/Python 和 Ruby API。当然，其他编程语言也有连接 MySQL 的 API，本章讲的只是比较流行的那些。以上每种 API 的相关小节，都会包含基本的 MySQL（或 MariaDB）连接教程，也会介绍如何使用该 API 发起数据库查询。

一般来说，掌握一种 API 便足够了，可能你也想跳到你所了解的（或你所用的）编程语言的那一节。我就比较喜欢 Perl 和 Perl DBI。Perl 很贴近自然语言（比如英语和意大利语）。而如果你没有什么特别喜好，只想学习一种 API，那么可以试试 PHP API，它带有很多用于与 MySQL 交互的方法，很多人都用它，而且确实很容易掌握。你可以在网页中使用它的代码片段，或使用 Wordpress 和 Drupal 之类的内容管理系统。

不过本书不会教你如何使用任何一种编程语言，因为我觉得你应该能通过其他书本或在线资源来学习它们的基本用法。我们只会涉及用这些语言来连接数据库的基本写法。

在阅读以下任何一节之前，你都应该先创建示例和习题所需的 API 用户账号。而最后的习题没有特别要求使用哪种 API，可以哪种顺手就用哪种。

16.1　创建API用户账号

假设我们编写的 API 程序最终是开放给外界使用的，那么，我们就得专门创建一个用户

账号（创建用户账号在第 13 章讲过）。我们可以给它起名为 public_api，并只授予它在 `rookery` 和 `birdwatchers` 数据库中的 `SELECT` 权限。执行以下命令：

```
CREATE USER 'public_api'@'localhost'
IDENTIFIED BY 'pwd_123';

GRANT SELECT
ON rookery.*
TO 'public_api'@'localhost';

GRANT SELECT
ON birdwatchers.*
TO 'public_api'@'localhost';
```

这就创建了密码为 pwd_123 的用户账号 public_api@localhost。你可以设置一个安全性更高的密码。这个账号只能从 localhost 访问两个数据库。具体来说，它只能执行 `SELECT`，不能删改任何数据。我们会在将要创建的 API 程序中使用它，这些 API 程序通过公众网页来检索数据。

此外，对于我们将要编写的某些程序来说，还需要一个管理员账号 admin_members，用来管理网站的会员信息。我们用以下语句来创建这个账号：

```
CREATE USER 'admin_members'@'localhost'
IDENTIFIED BY 'doc_killdeer_123';

GRANT SELECT, UPDATE, DELETE
ON birdwatchers.*
TO 'admin_members'@'localhost';
```

这个管理员账号只可以在 `birdwatchers` 数据库中选取、更新和删除数据。其实我们主要就是用它来操作 `humans` 表，而只是偶尔用它来操作该数据库中的其他表。因为它与 `rookery` 数据库无关，所以我们没让它在 `rookery` 中有任何权限。

16.2 C API

虽然现在 C 语言没有像以前那么流行了，但它仍然强大。事实上，MySQL 的核心软件就是用 C 写的，而 C API 也是 MySQL 提供的。本节就来简单地讲讲如何使用 C 和 C API 来连接和查询数据库，这会包括一些需要你必知必会的 C API 的基本组件和常规写法。

16.2.1 连接MySQL

使用 C 与 MySQL 交互时，我们需要先准备好一些变量，这些变量会在数据库连接时存储数据，以及我们执行查询的结果。然后，我们需要连接服务器。为了简单起见，我们会引入两个 C 头文件：带有基本 C 函数和变量的 stdio.h，以及带有 MySQL 特定函数和定义的 mysql.h（这两个文件分别来自 C 和 MySQL 或 MariaDB；如果你的服务器有 C 和 MySQL，那就不需要另外下载这两个文件）。

```
#include <stdio.h>
#include "/usr/include/mysql/mysql.h"
```

```
int main(int argc, char *argv[ ])
{
   MYSQL *mysql;
   MYSQL_RES *result;
   MYSQL_ROW row;
...
```

stdio.h 两侧的 < 和 >，意思是叫 C 到默认位置（如 /usr/include）或用户的路径去查找 C 头文件。而因为 mysql.h 不在默认位置，所以要在双引号中写出绝对路径。你也可以写成 <mysql/mysql.h>，因为 C 头文件就在默认位置的子目录中。

然后，在 main 函数的开头，我们先准备用于连接 MySQL 的一些变量。第一行创建了一个指针，指向 MySQL 结构（它存储在 mysql 变量中）。而下一行则根据 mysql.h 提供的 MYSQL_RES 的定义，定义和命名了一个结果集。结果集要存储在 result 数组中，它将是一个包含行的数组。第三行使用 MYSQL_ROW 的定义，创建了行变量，后面会用来保存列的数组。

引入了头文件并定义了变量之后，就可以用 mysql_init 在内存中创建一个对象，用来与 MySQL 服务器进行交互：

```
...
if(mysql_init(mysql) == NULL) {
   fprintf(stderr, "Cannot Initialize MySQL");
   return 1;
}
...
```

这里的 if 语句是用来检查能不能初始化 MySQL 对象。如果初始化失败，就会打印一条信息，并结束程序。mysql_init() 会用我们在 main 开头定义的 MYSQL 结构来初始化 MySQL 对象（起名为 mysql 只是习惯而已）。如果 C 成功地初始化对象，那它就会进行下一步，尝试连接 MySQL 服务器：

```
...
if(!mysql_real_connect(mysql,"localhost",
  "public_api","pwd_123","rookery",0,NULL,0))
{
   fprintf(stderr, "%d: %s \n", mysql_errno(mysql), mysql_error(mysql));
   return 1;
}
...
```

从此例你应该很容易就能看出 mysql_real_connect 需要些什么参数：MySQL 对象的引用、主机名或 IP 地址、用户名和密码，以及所要操作的数据库。而此例用的账号就是本章开头创建的 public_api@localhost。剩下的三个参数是端口号、套接字文件的名字和客户端标识。如果你没有特别要求，那就填 0 和 NULL，让 mysql_real_connect 按默认的设置。

如果程序无法连接，那它就会把服务器报的错打印到标准错误流，并将错误号和错误信息按 %d: %s \n 来格式化。其中错误号（%d）可从 mysql_errno() 获取，错误信息（%s）则可从 mysql_error() 获取。而如果程序能连接，且没有报错，那么 mysql_real_connect 就会返回 1 以示成功，我们就可以进入下一环节。

16.2.2 查询MySQL

目前，我们只建立好了连接。还需要看看如何通过 C API 来执行 SQL 语句。

如果 API 程序已经连接 MySQL，那它就可以用查询函数（比如 mysql_query()）来发起查询。以下便是用 SELECT 来获取 birds 表的数据，并将结果展示出来的代码：

```
...

if(mysql_query(mysql,"SELECT common_name, scientific_name FROM birds")) {
   fprintf(stderr, "%d: %s\n",
   mysql_errno(mysql), mysql_error(mysql));
}
else {
   result = mysql_store_result(mysql);
   while(row = mysql_fetch_row(result)){
      printf("\%s - \%s \n", row[0], row[1]);
   }
   mysql_free_result(result);
}
mysql_close(mysql);
return 0;
}
```

此例的 if 语句用了 mysql_query() 来发起查询，但你也可以用 mysql_real_query()。后者能获取二进制数据，这样更安全。不过这个例子很简单，所以用 mysql_query() 就可以了。这个函数会在查询成功时返回 0，失败时返回非零值。所以，如果失败，就会打印出错误信息。如果成功，返回的 0 就会使程序忽略 if，而执行 else 语句块。

而在 else 语句块中，第一行用 mysql_store_result() 将查询的结果集存储在 result 变量中。

接着，在释放数据之前，用 while 循环遍历结果集的每一行。这里用到了 mysql_fetch_row()，使每一次循环查到的行都暂存在 row 变量中。因为我们能预估到这个 SELECT 会查出些什么列，所以可以直接在 printf 中定好格式来展示每一列。注意，抽取列的内容就与抽取数组元素是一样的写法（即 array [n]）。

而作为这个 else 语句块的收尾，在遍历过后，我们会用 mysql_free_result() 来释放 result 所指的内存。

最后，用 mysql_close() 来断开与 MySQL 的连接，并用右花括号来结束 main。

16.2.3 完整的最小C API程序

像我刚才那样分块来解释程序的组件，听起来会容易一点，但可能不利于对整体布局的掌握。所以，下面我再给出完整的版本。

```
#include <stdio.h>
#include "/usr/include/mysql/mysql.h"
int main(int argc, char *argv[ ])
{
   MYSQL *mysql;
```

```
    MYSQL_RES *result;
    MYSQL_ROW row;

    if(mysql_init(mysql) == NULL) {
        fprintf(stderr, "Cannot Initialize MySQL");
        return 1;
    }

    if(!mysql_real_connect(mysql, "localhost", "public_api",
       "pwd_123", "rookery", 0, NULL, 0)) {
       fprintf(stderr, "%d: %s \n", mysql_errno(mysql), mysql_error(mysql));
       return 1;
    }

    if(mysql_query(mysql,"SELECT common_name, scientific_name FROM birds")) {
       fprintf(stderr, "%d: %s\n",
       mysql_errno(mysql), mysql_error(mysql));
    }
    else {
       result = mysql_store_result(mysql);

       while(row = mysql_fetch_row(result)) {
             printf("\%s - \%s \n", row[0], row[1]);
       }
       mysql_free_result(result);
    }
    mysql_close(mysql);
    return 0;
}
```

16.2.4　用GNU C编译器编译

你可以用任何编译器来编译以上程序，但我在这里只给出 GNU C 编译器（gcc）的示例。gcc 是免费软件，很多系统都自带。如果想进行编译和链接，可以输入如下命令：

```
gcc -c `mysql_config --cflags` mysql_c_prog.c
gcc -o mysql_c_prog mysql_c_prog.o `mysql_config --libs`
```

当编译器试图编译 mysql_c_prog.c 时，它会检查代码是否有语法错误。若有，它会停止编译并报错。若无，则会准备执行生成的编译程序 mysql_c_prog。

16.3　Perl DBI

用 Perl 来连接 MySQL 的最简单的方法，就是使用 Perl DBI 模块。而阅读本节的前提是，你对 Perl 有基本的了解。我们只会关注如何在 Perl 程序中连接 MySQL、执行 SQL 语句，以及获取数据，并不会谈及 Perl 本身的特性。这意味着本节的 Perl DBI 入门讲解是专门针对 Perl 程序员的。

本节的示例如下：假设我们想给管理员写一个程序，方便他获取会员信息，并更改其中某些人的会员资格失效日期。为此，我们会用专为管理会员信息而设的 admin_members 用户账号。我们在本章开头创建过此账号。

16.3.1 安装

Perl DBI 是 Perl 核心的一部分。你可以从 CPAN（http://www.cpan.org/）下载 Perl 和 DBI 模块。

如果你的服务器上已安装 Perl（其实大多数服务器都安装了），那么可以用以下命令来安装 DBI：

```
perl -MCPAN -e 'install DBI'
```

如果没有安装 Perl，可以使用安装包管理工具，比如 yum，来安装 DBI。如果用 yum 的话，需要先以 root 或文件系统管理员账号登录，然后执行以下命令。

```
yum install perl perl-mysql
```

16.3.2 连接MySQL

若想与 MySQL 交互，需要先用 DBI 来连接它。连接过程很简单，就是用以下几行代码：

```
#!/usr/bin/perl -w
use strict;

use DBI;

my $user = 'admin_members';
my $password = 'doc_killdeer_123';
my $host = 'localhost';
my $database = 'birdwatchers';

my $dbh = DBI->connect("DBI:mysql:$database:$host", $user, $password)
        || die "Could not connect to database: " . DBI->errstr;
...
```

头两行启动 Perl，并设为严格模式（即 use strict），以尽量减少出错。下一行调用 DBI 模块。然后我们创建与登录 MySQL 相关的一些变量。在后面被拆成两行的那条语句中，这些变量被用来建立 MySQL 数据库连接。获取到的连接，我们会用 $dbh 指向它。如果连接失败，就执行 die 部分。如果成功，则往下执行。

16.3.3 查询MySQL

只连接 MySQL 是没什么用的，我们还要执行 SQL 语句。任何 SQL 语句都可以通过 API 来执行。唯一要考虑的是，数据库账号本身是否有权限执行 SQL。换句话说，如果你的账号只能执行 SELECT，那么程序就只能执行 SELECT。下面，我们就来看看如何通过程序在 MySQL 中查询和插入数据。

1. 查询数据

紧接上例，让我们用 SELECT 从 humans 表中获取一些会员信息。我们允许这个 Perl 程序在命令行读取一个姓氏，并以它作为查询条件。例如，当程序的使用者输入 XXX 做参数时，程序便给他返回姓 XXX 的会员。而为了让查询更灵活，我们会在 WHERE 子句中用 LIKE。

大概就是这样：

```
...
my $search_parameter = shift;

my $sql_stmnt = "SELECT human_id,
                 CONCAT(name_first, SPACE(1), name_last) AS full_name,
                 membership_expiration
                 FROM humans
                 WHERE name_last LIKE ?";

my $sth = $dbh->prepare($sql_stmnt);

$sth->execute("%$search_parameter%");
...
```

首先，我们通过 `shift` 来获取命令行参数，并用变量 `$search_parameter` 来持有它。接着，创建存有 SQL 语句的变量 `$sql_stmnt`。但注意，`WHERE` 子句中并没有指定姓氏，而是写了一个问号。这个问号其实是一个占位符，之后执行语句时，我们才会给它代入实际的参数。使用占位符是一种安全策略。具体可以参考 16.7 节。

创建了 `$sql_stmnt` 变量之后，我们用 `$dbh` 的 `prepare()` 函数，在数据库构建这个查询，并以 `$sth` 指向构建的结果。最后，调用这个 `$sth` 的 `execute()` 方法，并传递 `$search_parameter`，来代入占位符的位置。如果要替换多个占位符，可以在 `execute()` 中以逗号分隔的方式列出对应的实际参数。

连接 MySQL 并发起查询之后，剩下要做的就是取出结果集的数据，然后把它们展示给管理员。我们可以在 `while` 循环中用 `fetchrow_array()` 这个一次获取一行结果的函数来做：

```
...
while(my($human_id,$full_name,$membership_expiration) = $sth->fetchrow_array())
{
   print "$full_name ($human_id) - $membership_expiration \n";
}

$sth->finish();
$dbh->disconnect();
```

只要结果集的记录未被取尽，`while` 块中的代码就会被反复执行。在这里，我们将每行的两列数据存到 `$common_name` 和 `$scientific_name` 这两个变量中（每次循环都会重写这些变量），然后将它们连同换行符打印出来。

而倒数第二行则是用 `$sth` 的 `finish()` 来终结这个构建出的语句。最后一行用 `$dbh` 的 `disconnect()` 来断开数据库连接。当然，你也可以先不断开连接，继续在这个连接上创建并执行其他 SQL 语句。

从 MySQL 中获取数据的一个更好的做法，就是在 Perl 程序中将所有数据保存起来供以后使用，并在处理结果前关闭 MySQL 连接。打开 MySQL 的同时逐行获取数据会拖慢程序，这在数据量很大时尤为明显。因此，一次就获取所有行，并在内存中用二维数组来暂存，等到用时直接从该数组中取，可能会更高效。具体的做法就是用 `fetchall_arrayref()` 方法。它会给你构建这样的数组，并返回该数组的起始位置：

```
...
my $members = $sth->fetchall_arrayref();

$sth->finish();

foreach my $member (@$members){
   my ($human_id, $full_name, $membership_expiration) = @$member;
   print "$full_name ($human_id) - $membership_expiration \n";
}

$dbh->disconnect();
```

fetchall_arrayref() 获取所有行，把它们保存在内存的数组中，然后返回它的起始位置。我们把起始位置保存在 $members 中，然后用 foreach 循环地将 @$members 的每个数组赋给 $member，再在 foreach 块中，把 $member 数组的元素分别赋给 $human_id、$full_name 和 $membership_expiration。接着用 print 将它们打印出来。

注意，在 foreach 之前，我们就已经调用了 finish() 来结束这个查询语句，并让 MySQL 释放资源。如果你不需要做其他查询，那么甚至可以在 finish() 之后立即执行 disconnect()。这对后面的 foreach 是没有影响的，因为 fetchall_arrayref() 已将所有结果导出。

2. 更新数据

我们已经在前面的例子中讲了如何从表中查询数据，下面来看看更新数据的例子。我们要修改 $sql_statement 的内容，使其包含 UPDATE 语句，用来更新 humans 表的 membership_expiration 列。做法如下：

```
...
my ($human_id, $membership_expiration) = (shift, shift);

$sql_stmnt = "UPDATE humans
              SET membership_expiration = ?
              WHERE human_id = ?";

$sth = $dbh->prepare($sql_stmnt);
$sth->execute($membership_expiration,$human_id);
...
```

先使用 shift 两次，将用户输入的两个参数放到 $human_id 和 $membership_expiration 中。然后，编写一条带有两个占位符的 SQL 语句。接着，在构建出的语句上（$sth），调用 execute() 方法，按顺序传入刚才的两个变量，代入那两个占位符。

结果会更新 humans 表中 $human_id 所指的那行的 membership_expiration 列。因为 humans 表的 UPDATE 权限一般是不公开给外界的，所以，你最好还是限定内部 IP 带密码登录，才能使用该程序。

16.3.4 Perl DBI完整示例

分块来解释一个程序，听起来会容易一点，但可能不利于对整体布局的掌握。所以，我将刚才的代码片段都组合起来，创建一个程序，取名 member_adjust_expiration.plx。如下所示：

```
#!/usr/bin/perl -w use strict;

use DBI;

my $search_parameter = shift || '';
my $human_id = shift || '';
my $membership_expiration = shift || '';

my $user = 'admin_members';
my $password = 'doc_killdeer_123';
my $host = 'localhost';
my $database = 'birdwatchers';

my $dbh = DBI->connect("DBI:mysql:$database:$host", $user, $password)
        || die "Could not connect to database: " . DBI->errstr;

if($search_parameter && !$membership_expiration) {
   my $sql_stmnt = "SELECT human_id,
                    CONCAT(name_first, SPACE(1), name_last) AS full_name,
                    membership_expiration
                    FROM humans
                    WHERE name_last LIKE ?";

   my $sth = $dbh->prepare($sql_stmnt);
   $sth->execute("%$search_parameter%");

   my $members = $sth->fetchall_arrayref();

   $sth->finish();

   print "List of Members - '$search_parameter' \n";

   foreach my $member (@$members){
      my ($human_id, $full_name, $membership_expiration) = @$member;
      print "$full_name ($human_id) - $membership_expiration \n";
   }

}

if($human_id && $membership_expiration) {
   $sth = $dbh->prepare($sql_stmnt);
   $sql_stmnt = "UPDATE humans
                 SET membership_expiration = ?
                 WHERE human_id = ?";

   $sth = $dbh->prepare($sql_stmnt);
   my ($rc) = $sth->execute($email_address,$human_id);

   $sth->finish();

   if($rc) {
     print "Membership Expiration Changed. \n";
   }
   else {
     print "Unable to change Membership Expiration. \n";
```

```
    }
}

$dbh->disconnect();
exit();
```

在命令行执行该程序时，如果在程序的名字后面加上 Hollar 作为参数，那么它就会返回 Lexi Hollar 这个名字，后跟一个含有她的 `human_id` 的括号对，以及其会员资格失效日期。以下便是查询 Hollar 的做法与结果：

```
member_adjust_expiration.plx Hollar

List of Members - 'Hollar'
Lexi Hollar (4) - 2013-09-22
```

你可以再执行一遍这个程序，并加上一个新的失效日期，如下：

```
member_adjust_expiration.plx Hollar 4 2015-06-30
```

注意，要更新失效日期，你需要给三个参数。如果程序只收到一个参数（会员的姓氏），那么它会执行 `SELECT`，并显示用户信息；而如果收到三个参数，才会执行 `UPDATE`。当然，参数的顺序和格式要正确，才能完成 `UPDATE`。如果是 `UPDATE`，程序也会告诉你更新失效日期是否成功。

你可以把这个程序写得更精细。例如，允许用户选择日期、月数或年数，用 MySQL 的时间日期函数将其添加到失效日期中。你还可以加上 CGI Perl 模块，使用户能通过网页点击来调用它。而我们这个程序是很初级的，只是让你了解如何着手写一个 Perl API 来与 MySQL 交互。

16.3.5 更多信息

如果想学习 Perl，可以参考由 Randal Schwartz、brian d foy 和 Tom Phoenix 合著的《Perl 语言入门》（http://shop.oreilly.com/product/0636920018452.do）。想了解 Perl DBI 的更多相关知识，可以参考由 Alligator Descartes 和 Tim Bunce 合著的《Perl DBI 编程》（http://shop.oreilly.com/product/9781565926998.do）。想了解 Perl 引用和其他更高级的话题，可以参考 Randal Schwartz 写的《Perl 进阶》（http://shop.oreilly.com/product/0636920012689.do）。

16.4 PHP API

PHP 和 MySQL 是 Web 最流行的编程语言与数据库引擎的组合之一。其中的原因有很多，不过最主要的是它们都很高效、稳定和简单。此外，PHP 脚本还很容易与 HTML 搭配来生成网页。所以，本节就来试试在单个网页中用 PHP API 连接和查询 MySQL。

16.4.1 安装与配置

通过 PHP 来连接 MySQL 的 API，比较流行的有三个。而我建议你用 `mysqli`（MySQL Improved 的缩写）扩展，它是 `mysql` 扩展的新版。本节的例子也会用它来写。

大多数的 Linux 系统都预装了 PHP。而如果你要自己安装 PHP 和 mysqli 的话，可以使用 yum 之类的安装包管理工具。可以这样来做：

```
yum install php php-mysql
```

PHP 能嵌在网页中，这是它的一个很不错的特性。如果你要这样写，就可能会需要调一下网页服务器的配置。对于 Apache 来说，要在 Apache 配置文件中加一个 AddType，告诉网页服务器用 PHP 来执行网页文件中的代码。可以把下面这行放在 httpd.conf 中，以造成全局的影响，也可以把它放在含有 PHP 代码的 HTML 网页所在的目录的 .htaccess 文件中。

```
AddType application/x-httpd-php .html
```

修改 httpd.conf 的指令，需要重启 Apache 才能生效。而写在 .htaccess 中则不用这么做。

要将 PHP 与 MySQL 搭配使用，你可能还需要用 --with-mysql=/path_to_mysql 选项来配置 PHP。而在使用 yum 安装 PHP API 的情况中，这是不用的。

16.4.2 连接MySQL

要用 PHP 来与 MySQL 交互，得先连接 MySQL，以建立一个 MySQL 客户端会话。需要用到的代码不多，如下所示：

```
<?php

  $host = 'localhost';
  $user = 'public_api';
  $pw = 'pwd_123';
  $db = 'rookery';

  $connect = new mysqli($host, $user, $pw, $db);

  if (mysqli_connect_errno()) {
       printf("Connect failed: %s\n", mysqli_connect_error());
       exit();
  }

?>
```

这些代码是用 <?php...?> 包起来的，这样便可以嵌到 HTML 中。而如果你想写一个从命令行执行的程序，则要以 #!/usr/bin/php 开头。不过，此例是在网页中写代码。

这些代码首先创建了一些连接 MySQL 和选取默认数据库所需的变量，在这些变量之后，我们用 mysqli() 函数来建立连接，并用 $connect 变量指向建好的连接。如果连接没建成功，就打印错误信息，并停止脚本。如果成功，下一步就可以发起查询。其间连接会一直开着，直到最后我们才会关闭它。

16.4.3 查询MySQL

下面要做的便是在脚本中查询 birds 表，获取一些鸟的信息。你可以把以下代码接到之前建立连接的代码后面，不过要在同一个网页内。这样就能发起查询，取出 birds 表的行，

并把它们展示给用户：

```
<?php
  $sql_stmnt = "SELECT common_name, scientific_name
               FROM birds
               WHERE LOWER(common_name) LIKE LOWER(?)";
  $sth = $connect->prepare($sql_stmnt);

  $search_parameter = $_REQUEST['birdname'];
  $search_parameter = "%" . $search_parameter . "%";

  $sth->bind_param('s', $search_parameter);

  $sth->execute();
  $sth->bind_result($common_name, $scientific_name);

while( $sth->fetch() ) {
  print "$common_name - <i>$scientific_name</i><br/>";
}

$sth->close();
$connect->close();
?>
```

首先创建变量 `$sql_stmnt`，在其中保存一条想要执行的 SQL 语句。然后，用 `$connect` 的 `prepare()` 函数解析它，并以 `$sth` 指向解析出的执行计划。

这个脚本的使用方法是：用户在网址末尾带上自己的查询条件，然后发送请求。例如，在网址中加上 `?birdname=Avocet` 来发起查询，用户就可以收到一堆 Avocet 鸟的信息了。

Web 表单

Web 用户一般不会在网址结尾处输入变量名和搜索值。我们需要在这个正在建立的网页前面，提供另一个含有 HTML 表单的网页给用户，让他们在里面填查询参数。这个 Web 表单大致如下：

```
<h3>Search Birds Database</h3>
<form action="birds.html" method="post">
<p>Enter a parameter by which to search
the common names of birds in our database:</p>
<input type="text" name="birdname" />
<input type="submit" />
</form>
```

由这个表单来调用 PHP 网页，以正确的格式传递查询参数。

接下来的两行将查询参数取出，放到变量 `$search_parameter` 中。而因为我们的 SQL 语句用了 `LIKE`，所以还得给变量的前后拼接上 `%`。

之后，用 `bind_param()` 方法将参数与 `$search_parameter` 绑定。因为该列是字符串类型，所以还要给 `bind_param()` 的第一个参数传 `'s'`。绑定之后，便可调用 `execute()` 来执行查询了。

接着，用 `bind_result()` 来指定查询结果要赋到什么变量中。再用 `while` 循环调用 `$sth` 的 `fetch()` 方法，并把每次取到的值与 HTML 标签拼起来，再打印出来。最后，关闭执行计划和数据库连接。

脚本的运行结果就是根据查询条件，将查得的每种鸟一行行地展示出来。整个例子其实很简单，只用了几个 PHP 函数来获取并展示数据。以下是嵌入 HTML 后的完整代码：

```
<html>
<body>

<?php
  $search_parameter = $_REQUEST['birdname'];
  $host = 'localhost';
  $user = 'public_api';
  $pw = 'pwd_123';
  $db = 'rookery';

  $connect = new mysqli($host, $user, $pw, $db);

  if (mysqli_connect_errno()) {
       printf("Connect failed: %s\n", mysqli_connect_error());
       exit();
  }
?>

<h3>Birds - <?php echo $search_parameter ?></h3>
<p>Below is a list of birds in our database based on your search criteria:</p>

<?php
  $sql_stmnt = "SELECT common_name, scientific_name
              FROM birds
              WHERE common_name LIKE ?";
  $sth = $connect->prepare($sql_stmnt);

  $search_parameter = "%" . $search_parameter . "%";
  $sth->bind_param('s', $search_parameter);
  $sth->execute();
  $sth->bind_result($common_name, $scientific_name);

while($sth->fetch()) {
  print "$common_name - <i>$scientific_name</i><br/>";
}

$sth->close();
$connect->close();
?>

</body>
</html>
```

这个例子看起来与前两节的差不多。我们在代码的前后加了 `body` 和 `html` 标签，并在两段 PHP 代码之间加了一些文本。我们还给某些语句的位置做了调整，不过这并没有影响整体流程。如果 Web 用户查的是 Avocet 鸟，那么就会得到以下文本。

```
Birds - "Avocet"
Below is a list of birds in our database based on your search criteria:

Pied Avocet - Recurvirostra avosetta
Red-necked Avocet - Recurvirostra novaehollandiae
Andean Avocet - Recurvirostra andina
American Avocet - Recurvirostra americana
Mountain Avocetbill - Opisthoprora euryptera
```

16.4.4 更多信息

想学习更多关于 `mysqli` 的知识，可以看看 PHP 的网站，那里有详尽的手册，还有一本 *MySQL Improved Extension* 手册（http://php.net/manual/en/book.mysqli.php）。此外，你可以读读 Robin Nixon 写的《PHP、MySQL 与 JavaScript 学习手册》（http://shop.oreilly.com/product/0636920036463.do），这本书也能帮助你深入了解如何在网页中使用 PHP 来操作 MySQL。

16.5 Python

要想在 Python 中操作 MySQL，可以用 MySQL Connector/Python。它是用 Python 写的，只用到 Python 的标准库，不需要其他 Python 模块，连 MySQL 客户端库也不需要。

16.5.1 安装

首先需要在服务器上安装 MySQL Connector/Python。若在 Linux 系统上安装，可以用 `yum` 之类的安装工具。虽然你的服务器可能自带了 Python 和 Python 库，但你也可以试试能否安装，以防万一。从命令行执行以下命令。

```
yum install python python-libs mysql-connector-python
```

本节使用的是 Python 2，它是目前 Linux 和 Mac 上的主流版本。而 Python 3 也正在流行起来，它与 Python 2 在语法上有点不同，具体可以参考相关文档。如果想用 Python 3，或者另一个用来连接 Python 和 MySQL 的库，只需稍微修改本节的示例即可。

安装好连接器后，就可以开始编写并运行一个 Python 程序，来连接 MySQL 和查询数据库。本节的示例假设有个需求：数据库管理员希望我们能提供一个程序，以便查询数据库账号及各账号的权限。下面我们就来实现它。

16.5.2 连接MySQL

要用 Python 查询数据库，首先得建立连接。下面是 Python 程序的开头部分：

```
#!/usr/bin/python

import mysql.connector
```

```
config = {
    'user': 'admin_granter',
    'password': 'avocet_123',
    'host': 'localhost',
    'database': 'rookery'
}

cnx = mysql.connector.connect(**config)
cur = cnx.cursor(buffered=True)
```

第一行指定命令行用 Python 来执行此脚本。接着，引入 MySQL Connector/Python，即 `mysql.connector`。然后，创建一个散列常量来存储账号、密码等登录信息。这里，我们用的用户账号是 admin_granter@localhost（在 13.3.4 节建的），因为它有权限执行 `SHOW GRANTS`，并查询存有账号信息的 `mysql` 数据库，这正好符合我们的要求。

最后两行用于建立连接。其中，第一行调用 MySQL Connector/Python 的 `connect()` 方法，并把刚才的 `config` 散列常量传进去，再以 `cnx` 变量保存所建的连接。第二行则是创建一个游标对象 `cur`，以备后面执行查询时使用。

16.5.3　查询MySQL

因为没有 `SHOW USERS` 语句，所以我们必须亲自查询 `mysql` 数据库的 `user` 表，来获取用户账号信息。为此，先建一个变量来保存需要执行的 `SELECT` 语句，然后用 `execute()` 执行它，如下所示：

```
sql_stmnt =  ("SELECT DISTINCT User, Host FROM mysql.db "
              "WHERE Db IN('rookery','birdwatchers') "
              "ORDER BY User, Host")

cur.execute(sql_stmnt)
```

为了适应书页的宽度，我们将 `SELECT` 语句拆成几行了。我们把变量传给 `execute()` 来执行该 SQL 语句。现在便可以取出每一行，解析每个域，再显示出来：

```
for row in cur.fetchall() :
  user_name = row[0]
  host_address = row[1]
  user_account = "'" + user_name + "'@'" + host_address + "'"

  print "%s@%s" % (user_name, host_address)

cur.close()
cnx.close()
```

对于 `cur` 的 `fetchall()` 方法所返回的结果集，我们用 `for` 来遍历它。这将在每次循环中，把每行的各个域都存在数组 `row` 中。而在 `for` 块中，我们再将各域取出，暂存到字符串变量 `user_name` 和 `host_address` 中。接着，用一些额外的文本跟它们拼接，以构造得好看一点，再把它们存到变量 `user_account` 中，它的内容类似 `lena_stankoska@localhost`。

逐行打印完 `user_account`，把结果展示给管理员，程序就结束了。然后，关闭游标对象和

MySQL 连接。

16.5.4 Python程序示例

像刚才那样把程序拆分，会比较容易说明，但可能让人不明白整个流程是怎样的。所以，下面我将那些代码片段进行合并，不过在里面加入了一些新内容，使程序更精细：

```
#!/usr/bin/python

import re
import mysql.connector

# connect to mysql
config = {
    'user': 'admin_granter',
    'password': 'avocet_123',
    'host': 'localhost',
    'database': 'rookery'
}

cnx = mysql.connector.connect(**config)
cur = cnx.cursor(buffered=True)

# query mysql database for list of user accounts
sql_stmnt = "SELECT DISTINCT User, Host FROM mysql.db "
sql_stmnt += "WHERE Db IN('rookery','birdwatchers') "
sql_stmnt += "ORDER BY User, Host"

cur.execute(sql_stmnt)

# loop through list of user accounts
for user_accounts in cur.fetchall() :
  user_name = user_accounts[0]
  host_address = user_accounts[1]
  user_account = "'" + user_name + "'@'" + host_address + "'"

  # display user account heading
  print "\nUser Account: %s@%s" % (user_name, host_address)
  print "------------------------------------------"

  # query mysql for grants for user account
  sql_stmnt = "SHOW GRANTS FOR " + user_account
  cur.execute(sql_stmnt)

  # loop through grant entries for user account
  for grants in cur.fetchall() :
    # skip 'usage' entry
    if re.search('USAGE', grants[0]) :
      continue

    # extract name of database and table
    dbtb = re.search('ON\s(.*)\.+?(.+?)\sTO', grants[0])
    db = dbtb.group(1)
    tb = dbtb.group(2)
```

```
        # change wildcard for tables to 'all'
        if re.search('\*', tb) :
          tb = "all"

        # display database and table name for privileges
        print "database: %s; table: %s" % (db,tb)

        # extract and display privileges for user account
        # for database and table
        privs = re.search('GRANT\s(.+?)\sON', grants[0])
        print "privileges: %s \n" % (privs.group(1))

    cur.close()
    cnx.close()
```

很明显以上代码比之前的片段多了很多内容。为了便于理解，我加上了注释。接下来，我们就再分析一遍这个代码（特别是新加的部分）。

一开始，它先查出一些账号，把它们保存到 `user_accounts` 数组中。然后，通过 `for` 循环，遍历 `user_accounts` 的每一行，抽取每一个 `user_account`，并打印出每个用户名和主机作为标题。至此，它跟之前的代码都还是挺像的。

接着，对于每个账号，我们都在 `sql_stmnt` 中新加入 `SHOW GRANTS`，并再用一个 `for` 循环来遍历 `fetchall()` 的结果集（保存在 `grants` 变量中）。在遍历的过程中，若碰到某行含有 `USAGE`，则跳过该行。而对于那些没跳过的，则要解析出数据库名和表名，存到变量 `db` 和 `tb` 中，然后打印出来。最后两行抽取权限并打印出来。

以下是在我的系统上运行这个 Python 程序所得的部分结果：

```
User Account: lena_stankoska@localhost
------------------------------------------
database: `rookery`; table: all
privileges: SELECT, INSERT, UPDATE, DELETE

database: `birdwatchers`; table: all
privileges: SELECT, INSERT, UPDATE


User Account: public_api@localhost
------------------------------------------
database: `birdwatchers`; table: all
privileges: SELECT

database: `rookery`; table: all
privileges: SELECT
```

因为 MySQL 本身并没有内置函数可以做到这些，所以此程序非常便于管理员查看每个账号在哪些数据库和表上有些什么权限。

16.5.5 更多信息

如果想了解更多有关 MySQL Connector/Python 的内容，可以到 MySQL 网站查看详尽的手册，其中包括 *MySQL Connector/Python Developer Guide*（http://dev.mysql.com/doc/connector-python/en/）。还可以读读 Mark Lutz 写的《Python 语言入门》（http://shop.oreilly.com/product/0636920028154.do）。

16.6 Ruby API

Ruby 现在已经是一门非常流行的语言了。同样，它也可以用来创建操作数据库的程序。如果要操作 MySQL，我们有两个选择。其中，以 MySQL 的 C API 构建的 MySQL/Ruby 模块因为拥有与 C API 一样的方法，所以特别适合那些熟悉 C API 的人。而另一个 Ruby/MySQL 模块（注意，它的名字跟刚才那个相比是倒转的），是用 Ruby 写的，Ruby on Rails 也采用它。不过本节的例子只用 MySQL/Ruby 模块。

16.6.1 安装和准备使用MySQL/Ruby

在写与 MySQL 交互的 Ruby 程序之前，我们先来安装 MySQL/Ruby 模块。它和 MySQL C API 使用的函数相同。在 Linux 上，可以用 `yum` 之类的安装包管理工具来做。先以 root 或其他管理员账号登录系统，然后在命令行中执行以下命令：

```
yum install ruby ruby-mysql
```

如果你用不了 `yum`，可以试试到 MySQL 的网站去下载 Ruby 模块（http://dev.mysql.com/downloads/ruby.html），并按照指南进行安装。

安装好 Ruby 和 MySQL/Ruby 模块后，就可以开始编写并执行 Ruby 程序，来连接 MySQL 和查询数据库。本节的程序示例很简单。我们要用 13.2.1 节所建的 admin_backup@localhost，到 `server_admin` 数据库去查询和插入数据。其中，`backup_policies` 表用来记录一些备份策略的相关信息。该数据库用于保存一些备份计划和实施情况。

先来创建 `server_admin` 数据库和 `backup_policies` 表：

```
CREATE DATABASE server_admin;

CREATE TABLE backup_policies
(policy_id INT AUTO_INCREMENT KEY,
backup_name VARCHAR(100),
file_format_prefix VARCHAR(25),
frequency ENUM('daily','weekly'),
days ENUM('first','every'), start_time TIME,
secure TINYINT DEFAULT 0,
location ENUM('on-site','off-site','both'),
tables_include VARCHAR(255) );
```

建好后，把表 14-2 中与备份策略相关的数据插入 `backup_policies` 表。执行以下 `INSERT` 语句：

```
INSERT INTO backup_policies
(backup_name, file_format_prefix, frequency,
 days, start_time, secure, location, tables_include)
VALUES
('rookery - full back-up', 'rookery-', 2, 1, '08:00:00', 0, 2, "all tables"),
('rookery - bird classification', 'rookery-class-', 1, 2, '09:00:00', 0, 1,
 "birds, bird_families, bird_orders"),
('birdwatchers - full back-up',
 'birdwatchers-', 2, 1, '08:30:00', 1, 2, "all tables"),
('birdwatchers - people', 'birdwatchers-people-', 1, 2, '09:30:00', 1, 1,
 "humans, birder_families, birding_events_children"),
('birdwatchers - activities', 'birdwatchers-activities-', 1, 2, '10:00:00', 0, 1,
 "bird_sightings, birding_events, bird_identification_tests,
  prize_winners, surveys, survey_answers, survey_questions");
```

此外，我们还需要在 `server_admin` 数据库中创建另一个表。我们把这个表叫作 `backup_reports`，以保存备份程序所生成的报告。创建这个表的 SQL 语句如下：

```
CREATE TABLE backup_reports
(report_id INT AUTO_INCREMENT KEY,
 report_date DATETIME,
 admin_name VARCHAR(100),
 report TEXT);
```

该表很简单，仅含有报告 ID、报告日期、生成报告的管理员名字，以及报告内容（即由我们接下来要写的备份程序所输出的内容）。因为我们将用 admin_backup 这个账号，所以还要给它授予操作 `server_admin` 数据库的权限。为此，需要执行以下 SQL 语句：

```
GRANT SELECT, INSERT ON server_admin.*
TO 'admin_backup'@'localhost';
```

这样，我们就可以开始写备份程序了。

16.6.2 连接MySQL

用 Ruby 查询数据库之前，必须先建立连接。所以，我们会这样写程序的开头部分：

```
require 'mysql'

user = 'admin_backup'
password = 'its_password_123'
host = 'localhost'
database = 'server_admin'

begin
    con = Mysql.new host, user, password, database

# Database Queries Here
# ...

rescue Mysql::Error => e
    puts e.errno
    puts e.error
```

```
ensure
    con.close if con
end
```

此片段展示了如何使用 Ruby 来连接 MySQL 并随后断开连接。首先，第一行是调用 Ruby 来解释此脚本的常规写法。接着，下一行引入 `mysql` 模块。然后，再定义一些与连接相关的变量（你不一定要起上面那样的名字）。

接着，把所有与数据库交互的动作都放到 `begin` 块中。在该块中，先用刚才定义的变量建立一个数据库连接（这些变量或参数的顺序必须如上所示）。

建立好连接后，就可以执行 SQL 语句了。不过，我在这里暂时把查询数据库的内容省略掉了，之后再详细写出。

如果建立不了连接，就用后面的 `rescue` 块来接管报错，在里面用 `puts` 把错误信息打印出来。而最后的 `ensure` 块及其内容，则用来确保程序在结束时，关掉打开的连接。

16.6.3 查询MySQL

在试过运行一个简单的 Ruby 程序并与 MySQL 连接和断开连接之后，我们继续看看如何在连接开启的情况下，用 Ruby API 来查询数据库。

先简单地尝试从 `birds` 表中查询一些 Avocet 鸟。为此，我们先要定义一个变量，用于保存我们想要执行的 `SELECT` 语句。然后，将它交给 `query()` 方法去执行。这部分的程序如下所示：

```
sql = "SELECT common_name, scientific_name
       FROM birds
       WHERE common_name LIKE '%Avocet%'"

rows = con.query(sql)

rows.each do |row|
  common_name = row[0]
  scientific_name = row[1]
  puts common_name + ' - ' + scientific_name
end
```

`query()` 执行之后，我们在其返回结果上调用了 `each`，来遍历查得的所有行。在遍历过程中，每经过一行，就将该行赋给 `row` 数组。然后，因为每行数据都已包装成数组的形式，所以，我们就以获取数组元素的方式，将其位置 0 和 1 的对象，分别赋给 `common_name` 和 `scientific_name`。最后，将这两个变量的内容以横杠拼接，打印出来。

16.6.4 MySQL/Ruby程序示例

拆分代码会比较容易说明，但可能让人搞不清楚整个流程。所以，下面我提供一个用到 MySQL/Ruby 模块的完整 Ruby 程序。这个程序的用途与刚才的不同，它是用来根据制定好的备份策略检查备份文件的目录（这个任务在 14.3 节中介绍过）。它可以向管理员展示过去几天所产生的备份文件，并将这些结果的报告保存到 `server_admin` 数据库的 `backup_`

reports 表中：

```
#!/usr/bin/ruby

require 'mysql'

# create date variables
time = Time.new
yr = time.strftime("%Y")
mn = time.strftime("%m")
mon = time.strftime("%b")
dy = time.strftime("%d")

# variables for connecting to mysql
user = 'admin_backup'
password = 'its_password_123'
host =  'localhost'
database = 'server_admin'

# create other initial variables
bu_dir = "/data/backup/rookery/"
admin_name = "Lena Stankoska"

bu_report =  "Back-Up File Report\n"
bu_report += "-----------------------------------------------------\n"
puts bu_report

it = 0
num = 7

begin
   # connect to mysql and query database for back-up policies
   con = Mysql.new host, user, password, database
   sql = "SELECT policy_id, backup_name, frequency,
          tables_include, file_format_prefix
          FROM backup_policies"
   policies = con.query(sql)

   policies.each_hash do |policy|      # loop through each row, each policy
     # capture fields in variables
     bu_name = policy['backup_name']
     bu_pre = policy['file_format_prefix']
     bu_freq = policy['frequency']

     # assemble header for policy
     bu_header = "\n" + bu_name + " (performed " + bu_freq + ")\n"
     bu_header += "(" + bu_pre + "yyyy-mmm-dd.sql) \n"
     bu_header += "-----------------------------------------------------\n"
     bu_report += bu_header
     puts bu_header

     until it > num do          # iterate through 7 back-up files (i.e., days)
        bk_day = dy.to_i - it

        # assemble backup filename
```

```
            bu_file_suffix = yr + "-" + mon.downcase + "-" + bk_day.to_s + ".sql"
            bu_file = bu_pre + bu_file_suffix
            bu_path_file = bu_dir + bu_file

            # get info. on back-up file if it exists
            if File::exists?(bu_path_file)
               bu_size = File.size?(bu_path_file)
               bu_size_human = bu_size / 1024

               bu_file_entry = bu_file + " (" + bu_size_human.to_s + "k)"
               bu_report += bu_file_entry + "\n"
               puts bu_file_entry
            end
            it +=1
         end
         it = 0
      end
    end

    begin
        # insert report text accumulated in backup_reports table
        con = Mysql.new host, user, password, database
        sql = "INSERT INTO backup_reports
               (report_date, admin_name, report)
               VALUES (NOW(), ?, ?)"
        prep_sql = con.prepare sql
        prep_sql.execute(admin_name,bu_report)

    rescue Mysql::Error => e
        puts e.errno
        puts e.error

    ensure
        con.close if con
    end
```

虽然代码的每一部分都加了注释，但我还是想再大概说一下（不过其中某些地方会着重一点）。

首先，获取当前日期，并设置一些变量，用于查找备份文件。之所以按日期来检查，是因为在我们的备份策略中，备份文件的命名规范就是这样的。

然后，跳到 bu_report 那里，这个变量是用来存储报表的。报表的内容会随着程序的运行，不断地追加新的条目（这些条目也会在产生后即刻被打印到屏幕上）。最后，这个报表变量会代入 SQL 语句，被插到 backup_reports 里。

回过来看看第一个 begin 块。这里，我们先执行了一个 SELECT，来查询 backup_policies 表的备份策略。该表保存着各备份文件的命名规范（表名和日期后缀）。我们把这些备份策略保存在 policies 这个散列常量中。使用 each 遍历每一个 policy，然后每个都用 until 循环七次，拼接出最近七天的备份文件名。检查备份文件存放目录中是否真的存在这些文件名，如果存在，则将名字及文件大小记录到报表变量 bu_report 中。

第二个 begin 块执行一个 INSERT，把 bu_report 连同日期和管理员的名字录入 backup_

`reports` 表。以下便是该表的其中一条记录。

```
*************************** 62. row ***************************
  report_id: 62
report_date: 2014-10-20 14:32:37
 admin_name: Lena Stankoska
     report: Back-Up File Report
-----------------------------------------------------

rookery - full back-up (performed weekly)
(rookery-yyyy-mmm-dd.sql)
-----------------------------------------------------
rookery-2014-oct-20.sql (7476k)
rookery-2014-oct-13.sql (7474k)

rookery - bird classification (performed daily)
(rookery-class-yyyy-mmm-dd.sql)
-----------------------------------------------------
rookery-class-2014-oct-20.sql (2156k)
rookery-class-2014-oct-19.sql (2156k)
rookery-class-2014-oct-18.sql (2156k)
rookery-class-2014-oct-17.sql (2154k)
rookery-class-2014-oct-16.sql (2154k)
rookery-class-2014-oct-15.sql (2154k)
rookery-class-2014-oct-14.sql (2154k)
rookery-class-2014-oct-13.sql (2154k)
birdwatchers - full back-up (performed weekly)
(birdwatchers-yyyy-mmm-dd.sql)
-----------------------------------------------------
birdwatchers-2014-oct-20.sql (28k)
birdwatchers-2014-oct-13.sql (24k)

birdwatchers - people (performed daily)
(birdwatchers-people-yyyy-mmm-dd.sql)
-----------------------------------------------------
birdwatchers-people-2014-oct-20.sql (6k)
birdwatchers-people-2014-oct-19.sql (6k)
birdwatchers-people-2014-oct-18.sql (6k)
birdwatchers-people-2014-oct-17.sql (4k)
birdwatchers-people-2014-oct-16.sql (4k)
birdwatchers-people-2014-oct-15.sql (4k)
birdwatchers-people-2014-oct-14.sql (4k)
birdwatchers-people-2014-oct-13.sql (4k)

birdwatchers - activities (performed daily)
(birdwatchers-activities-yyyy-mmm-dd.sql)
-----------------------------------------------------
birdwatchers-activities-2014-oct-20.sql (15k)
birdwatchers-activities-2014-oct-19.sql (15k)
birdwatchers-activities-2014-oct-18.sql (15k)
birdwatchers-activities-2014-oct-17.sql (15k)
birdwatchers-activities-2014-oct-16.sql (15k)
birdwatchers-activities-2014-oct-15.sql (13k)
birdwatchers-activities-2014-oct-14.sql (13k)
birdwatchers-activities-2014-oct-13.sql (13k)
```

16.6.5　更多信息

如果想学习更多关于 Ruby 与 MySQL 交互的相关知识，可以看看 Tomita Masahiro 写的使用手册（http://www.tmtm.org/en/mysql/ruby/）。Tomita Masahiro 是 MySQL Ruby 模块的创造者。此外，Michael Fitzgerald 写的《学习 Ruby》（http://shop.oreilly.com/product/9780596529864.do）应该也会对你有所帮助。

16.7　SQL注入

凡是公开给他人使用的数据库 API，无论是 Web 还是其他形式，都有可能会被用来攻击数据库。通过 API 往返于用户界面和服务器之间的数据是可以被篡改的，尤其是黑客可以把 SQL 语句嵌入数据，从而让 API 拼接出恶意查询语句，进而破坏数据，窃取敏感或重要信息，或创建全能账号做进一步攻击。这就是人们所说的 SQL 注入。

这个问题其实与引号有点关系：要把 SQL 语句注入字符串参数，黑客只需要闭合引号，接上分号，再往后添加另一条自己写的 SQL 语句。而如果是想注入数字参数，那直接加一个不带引号的子句就可以另起 SQL 语句。

下面我们以 PHP API 为例，演示一下如何对没有占位符的 SQL 语句进行注入。假设我们的 SQL 语句中嵌入了一个名为 `$search_parameter` 的变量：

```
$sql_stmnt = "SELECT common_name, scientific_name
              FROM birds
              WHERE common_name LIKE '%$search_parameter%'"
```

假设黑客在使用 API 项目时，不传鸟的俗名，而是传以下这串内容（两头都带单引号）：

```
'; GRANT ALL PRIVILEGES ON *.* TO 'bad_guy'@'%'; '
```

那么就会拼接出如下 SQL 语句：

```
SELECT common_name, scientific_name FROM birds
WHERE common_name LIKE '%';

GRANT ALL PRIVILEGES ON *.* TO 'bad_guy'@'%';

'%';
```

它实际上是三条 SQL 语句，而不只是一条。其中，第一条语句会查出几乎所有的鸟。而第三条，因为不是完整的 SQL 语句，所以会报错。但这都不算大问题，第二条语句才最关键。系统会创建一个用户账号，并允许它不论从哪里登录，都可以无需密码全权操作所有数据库和所有表。如果 API 所使用的用户账号确实具有所有数据库和所有表的 `GRANT` 和 `ALL` 权限，那么该语句就会顺利执行，创建一个没有任何限制的账号 bad_guy，这是非常危险的。

为了避免 SQL 注入这种问题，我们应该在 API 中查询参数的位置上，使用占位符，而非字符串拼接。本章前面的例子使用过占位符。这种方法会将参数与语句独立开来，可以通过对参数中的引号进行跳脱处理来实现。这种方法可能看上去没什么了不起，但确实非常有效。

如果上例用了占位符，那么出来的语句就会是这样：

```
SELECT common_name, scientific_name FROM birds
WHERE common_name LIKE '%\';

GRANT ALL PRIVILEGES ON *.* TO \'bad_guy\'@\'%\';

'%';
```

由于黑客输入的引号都跳脱成普通字符，它们不会被当成是字符串的结束标志，因此，当遇到黑客输入的分号时，不会开始一条新的 SQL 语句。用这样的参数去查询是没有结果的，因为表中确实没有哪种鸟叫这样的名字。而更重要的是，用户账号 bad_guy 不会被创建。

16.8 小结

API 可以让不懂 MySQL 的用户也能创建程序，同时，也能隔绝用户对数据库的直接操作。它能给你提供一种更高层次的安全控管（尤其是控管来自 Web 的未知用户的访问）。另外，当 MySQL 本身没有用于查询你想要的信息的函数时，你可以写一个 API 程序来进行查询，补充 MySQL 的不足。总的来说，API 是自定义 MySQL 和 MariaDB 的强大工具。

本章的 API 程序示例涉及对数据的查询、插入和更新，其中有简单的，也有进阶的。我们几乎没有进行错误检查，而且只执行了一些简单的任务。虽然有些例子非常基础，但应该足够让你了解如何用 API 连接 MySQL 和 MariaDB，以及查询数据库。如果想做得更好，就要自己去好好学习相关的编程语言和 MySQL，并使用 API 提供的更多函数。关于这方面，也可以到每节的末尾，看看我给你推荐的书籍和其他资源。

16.9 习题

你可以用自己喜欢的任何语言的 API 来完成以下习题。如果没有特别的偏好，可以试试 PHP。它有很多用户，而且很容易上手。

(1) 写一个 API 程序，连接 MySQL，并查询 `rookery` 数据库。用 `SELECT` 获取一些鸟的信息。要求用 `JOIN`（可参考 9.2 节）来连接 `birds`、`bird_families` 和 `bird_orders` 这三个表，查出 `birds` 的 `bird_id`、`common_name` 和 `scientific_name`，以及 `bird_families` 和 `bird_orders` 的 `scientific_name`，并用 `LIMIT` 使其只返回 100 种鸟。写完后，执行看看。如果是用 PHP，可试试通过浏览器来显示结果。

(2) 写一个 API 程序，从命令行或浏览器（如果使用的是 PHP）接收程序用户的数据。让程序连接 MySQL 和 `birdwatchers` 数据库。让它执行 `INSERT`，来向 `humans` 表中添加用户提供的数据，注意只是添加到 `formal_title`、`name_first` 和 `name_last` 这几列里。用 `CURDATE()` 设置 `join_date`，然后在 `membership_type` 中填 basic。

写完后，运行它并输入几个虚构的人名，再用 `mysql` 客户端登录 MySQL，看看数据录入得是否正确。

(3) 登录 MySQL，用 `CREATE TABLE`（在 4.2 节中讲过）在本章开头创建的 `server_admin` 数据库中建表 `backup_logs`。表结构至少要有记录日期和时间，以及备份文件名称的列，其他随意。

使用 `GRANTS` 语句给 admin_backup 至少授予这个新表的 `INSERT` 和 `SELECT` 权限（可参考 13.2.2 节）。

在 14.1.4 节中，我们试过用 shell 来制作备份脚本。现在试试用 API 来做，使它能通过命令行（非浏览器）被调用，完成相同的任务。你不需要从头写出一个备份工具，只要让它调用 `mysqldump` 就可以了。写完后，运行程序看看能不能生成备份文件，以及文件名是否正确。如果本题超出你现在的水平，那就暂时先跳过，等以后你对 API 更熟练时，再回头来做。

如果 API 程序能生成正确的备份文件，那就增加一步，让它连接 MySQL，记录它已经成功运行。用 `INSERT` 插入一行，记录程序运行的日期和生成的备份文件的名称。改完后，再运行一次，看看有没有录入这些信息。

能记录的话，就加一行到 `cron` 或其他调度程序中，使你写的备份程序自动运行。加的时候，可以把时间设定在稍后，以便尽快检查有没有加成功。确认成功后，你可以按自己的意愿移除或保留该计划任务。

(4) 写一个 API 程序，先查出所有鸟科，供用户从中选择，然后列出所选鸟科的所有鸟种。如果你用的是可以在浏览器中使用的 PHP，那就给每个科加上链接，把科的信息发回 PHP 端，用于后续查询。

如果你的程序只能通过命令行调用，那就在显示科名时也显示 `family_id`。当用户要查某科的鸟种时，让其使用同一命令，但要带上 `family_id`。如果没有带上 `family_id`，就会显示一堆科名。反之，则会显示这一科的鸟种列表。写完后，运行程序测试一下。

关于作者

Russell J.T. Dyer 是住在意大利米兰的一位作家兼编辑。他现任 MariaDB 公司课程主管，并曾在 MySQL 公司做过六年的知识库编辑。他著有《MySQL 核心技术手册》一书（http://shop.oreilly.com/product/9780596514334.do），还写过数百篇文章，涉及的领域包括软件（特别是 MySQL）、旅游、摄影和经济。

在亚马逊网站，你还可以找到 Russell J.T. Dyer 写的第一本小说 *In Search of Kafka*。这本书的编辑也是 Andy Oram。小说描写的是一位计算机程序员不小心惹上了法律麻烦，读起来很有趣。想了解更多有关 Russell 及其作品的信息，可以浏览他的个人网站（http://russelljtdyer.com）。

关于封面

本书封面上的动物是黑带神仙鱼（banded angelfish），学名为弓背阿波鱼（Apolemichthys arcuatus）。所谓“黑带”，是指鱼身两侧从鱼眼延伸至鱼尾的黑带。跟其所属的盖刺鱼科（Pomacanthidae）中的其他海水神仙鱼一样，黑带神仙鱼两侧很扁，背鳍很软。另外，这种鱼也叫作“贼仙鱼”，它们居住在夏威夷和约翰逊环礁的中等深度水域的洞穴和礁石处。

不同品种的海水神仙鱼在行为习性上有很大的差异。在盖刺鱼科中，有的品种会组成一雄一雌的配偶，也有的会组成一条雄性与多条雌性相配的群体。而又因为它们是雌雄同体，并且雌性先熟，所以一旦唯一的雄性死去或离去，某条雌性就会转为雄性，继续维持鱼群的单雄多雌状态。

黑带神仙鱼主食海绵动物，此外也食藻类植物和某些无脊椎动物。因为这些食物很难养，所以想养黑带神仙鱼的水族馆很为难。一些商业水族馆为了维持养殖而去采挖珊瑚礁，导致人类潜水深度附近的珊瑚礁衰减。

O’Reilly 图书封面上的很多动物都濒临灭绝。它们都是自然界所剩无几的瑰宝。要想了解如何帮助它们，可以访问 http://animals.oreilly.com。

本书的封面图片来自 *Cuvier’s Animals*。

O'REILLY®

MySQL与MariaDB学习指南

也许你未曾意识到，但你时常在与MySQL或其分支打交道。作为高效且稳定的开源数据库，MySQL备受各大电商网站和社会媒体网站青睐。若想快速并深入了解如何使用和维护MySQL，本书便是绝佳参考。这本注重实践的学习指南以简单明了、条理清晰的方式，教你安装、使用和维护MySQL及其重要分支MariaDB。

本书作者既是MySQL与MariaDB专家，也是小说家。他以生动的语言和翔实的示例分析带你领略数据库设计和数据管理的方方面面。章末精心设计的习题将有助于你温故而知新。

- 创建和修改MySQL表，并在其中声明字段和列
- 通过示例，掌握数据的插入、选取、更新、删除、连接和子查询
- 使用字符串函数对列中的文本进行查找、抽取、格式化和转换
- 学习用于数学或统计运算，以及日期和时间格式化的相关函数
- 执行管理任务，例如管理账号、备份数据库和批量导入数据
- 使用PHP等各种编程语言的API连接和查询MySQL或MariaDB

“本书介绍的数据库开发和管理通用技能，将使你终身受益。”

——Monty Widenius
MySQL与MariaDB之父

“MySQL和MariaDB都是流行的数据库，阅读本书能让你快速地掌握它们。Russell举的例子简单易懂，并且贯穿整个学习过程，能助你迈入数据库专家行列。”

——Colin Charles
MariaDB公司前首席布道官，现任Percona公司首席布道官

Russell J.T. Dyer现任MariaDB公司课程主管，同时是一位小说家。他曾在MySQL公司做过近六年的知识库编辑，拥有丰富的MySQL实践经验，另著有《MySQL核心技术手册》。作为小说家，他目前正在创作第二部小说。

MYSQL DATABASE

封面设计：Ellie Volckhausen　马冬燕

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机／数据库／MySQL

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-43571-2

ISBN 978-7-115-43571-2
定价：79.00元